Towards Scientific Applications for Interactive Ray Casting

Matthias Groß

Vom Fachbereich Informatik der Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.) genehmigte Dissertation.

Dekan und Vorsitzender der Promotionskommission:
Prof. Dr. Karsten Berns
Berichterstatter:
Prof. Dr. Hans Hagen
Prof. Dr. Jörg Meyer

Tag der Wissenschaftlichen Aussprache:
7.7.2009

(D 386)

To my beloved wife

Dwi Retnani Poetranto Groß

for your love, support, and trust

Acknowledgements

This work would not have been possible without the support from many people. Thus, I want to acknowledge my appreciation to a number of persons for their contribution in this work. Many thanks to:

My PhD advisor Prof. Dr. Hans Hagen, for his support and trust. Whenever my way was stony, whenever I doubted my work, he showed confidence in me and gave me therewith the strength to continue.

My supervisor Dr. Franz-Josef Pfreundt, for his guidance and our fruitful discussions.

My colleague Andreas Wiegmann, for beeing highly supportive and giving valuable suggestions. The collaboration was really delightful.

My colleague and fellow Dr. Gerd Marmitt, for his scientific advice and for cheering me up whenever needed.

Prof. Jörg Meyer, for co-revising this thesis.

My institute ITWM, for the scholarship that allowed me to accomplish this thesis.

My dear parents, for their love, caring, and understanding. They always support and believe in me, whatever I do.

My colleagues, fellows and friends. Even if I don't mention your name one by one, I really appreciate your help and support.

A special thanks to my beloved wife. She gave me the final impulse for this work, supported me with valueable suggestions, and gave me strength even when we both went together through rough times.

Abstract

Interactive visualization of large structured and unstructured data sets is a permanent challenge for scientific visualization. Large data sets are for example created by magnetic resonance imaging (MRI), computed tomography (CT), Computational fluid dynamics (CFD) finite element method (FEM), and computer aided design (CAD).

For visualizing those data sets not only accelerated rasterization by means of using specialized hardware i.e. graphics cards is of interest, but also ray casting, as it is perfectly suited for scientific visualization. Ray casting does not only support many rendering modes (e.g., opaque rendering, semi transparent rendering, iso surface rendering, maximum intensity projection, x-ray, absorption emitter model, ...) for which it allows the creation of high quality images, but it also supports many primitives (e.g., not only triangles but also spheres, curved iso surfaces, NURBS, implicit functions, ...). It furthermore scales basically linear to the amount of processor cores used and - this makes it highly interesting for the visualization of large data sets - it scales for static scenes sublinear to data size.

Interactive ray casting is currently not widely used within the scientifc visualization community. This is mainly based on historical reasons, as just a few years ago no applicable interactive ray casters for commodity hardware did exist. Interactive scientific visualization has only been possible by using graphics cards or specialized and/or expensive hardware. The goal of this work is to broaden the possibilies for interactive scientific visualization, by showing that interactive CPU based ray casting is today feasible on commodity hardware and that it may efficiently be used together with GPU based rasterization.

In this thesis it is first shown that interactive CPU based ray casters may efficiently be integrated into already existing OpenGL frameworks. This is achieved through an OpenGL friendly interface that supports multiple threads and single instruction multiple data (SIMD) operations.

For the visualization of rectilinear (and not necessarily cartesian) grids are new implicit *k*d-trees introduced. They have fast construction times, low memory requirements, and allow on today's commodity desktop machines interactive iso surface ray tracing and maximum intensity projection of large scalar fields.

A new interactive SIMD ray tracing technique for large tetrahedral meshes is introduced. It is very portable and general and is therefore suited for portation upon different (future) hardware and for usage upon several applications.

The thesis ends with a real life commercial application which shows that CPU-based ray casting has already reached the state where it may outperform GPU-based rasterization for scientific visualization.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

This chapter gives a brief overview to different visualization techniques and methods, acceleration techniques and -structures for ray tracing, and shows why ray casting outperforms rasterization for large static scenes. The later is the motivation of this thesis, which tries to push interactive ray casting towards scientific applications.

1.2 Rendering Techniques

Rendering techniques are cathegorized into object order and image order techniques.

1.2.1 Object Order

Object order techniques iterate over the primitives and project them onto the image plane.

- **Rasterization** projects flat primitives (preferably triangles) onto the image plane (21).
- **Cell projection** projects volumetric primitives onto the image plane (29).

- **Vertex projection** (also called splatting) projects the primitives' vertices onto the image plane (56).
- **Shear warp** is restricted to cartesian grids, where it projects sheared slices of the grid onto an axis aligned plane and warps this projection onto the image plane (23).

1.2.2 Image Order

Image order techniques iterate over the image plane's pixels, where for each pixel the corresponding primitives are found which then define the pixel's color.

- **Ray casting** generates for each pixel a ray and shoots (casts) it into the scene, where it is tested for intersection against the scene's primitives (1). The corresponding pixel color is evaluated using a local illumination model.
- **Ray tracing** is the generalization of ray casting and also supports global illumination models by shooting (tracing) secondary rays such as shadow rays, reflected rays and refracted rays (57) through the scene.

1.3 Rendering Modes

The different rendering techniques display a scene's primitives by different rendering modes. Some of the here given render modes are commonly only applied to volume rendering.

- **Opaque Rendering** displays the primitives opaque. Only the nearest primitives are visibile as they occlude rear-lying primitives.
- **Semi Transparent Rendering** displays the primitives semi transparent, where some blend function defines the final transparency and color.
- **X-Ray** displays for each pixel the integration of density values reached at the primitives' points belonging to that pixel.
- **Maximum Intensity Projection (MIP)** displays for each pixel the highest density reached at the primitives' points belonging to that pixel.

- **Iso Surface Rendering (Iso Surfacing)** corresponds to rendering the iso surface S_{iso} to the iso value s_{iso} in some scalar field S: $S_{iso} = \{s \in S : s = s_{iso}\}$.

- **Semi Iso Surfacing** does not only render the iso surface, but also values greater/smaller than the iso value $s \geqq s_{iso}$ / $s \leqq s_{iso}$. Semi iso surfacing is therefore a combination of iso surfacing and opaque rendering (in fact, rendering $s \lessgtr s_{iso}$ means rendering S which corresponds to pure opaque rendering).

1.4 Acceleration Techniques for Ray Tracing

As the focus of this thesis is primarily on interactive ray casting a brief summary of different acceleration techniques for ray tracing is given here.

- **Hierarchical acceleration structures** are used to allocate the scene's primitive hierarchicaly. Rays traverse this acceleration structure, and only primitives belonging to the hierarchies lowest level elements (leaf nodes) are tested for intersection against the rays. This typically strongly minimizes the amount of ray primitive intersection tests for the low cost of traversing the hierarchy, leading typically to a major speed up compared to brute force ray tracing, where each ray is tested against each primitive.

- **Packet-based ray tracing** works on operating not on single rays, but on packets of multiple rays. The entire packet traverses the hierachical acceleration structure and is tested against the relevant primitives.

- **Single instruction multiple data (SIMD) support** allows either for one ray to traverse multiple parts of a hierarchical accceleration structure and to be tested agains multiple primitives, or for multiple rays (e.g., a ray packet) to traverse the hierarchical acceleration structure and to test them against primitives.

- **Early ray termination** stops tracing a ray further trough a scene if the corresponding pixel color will not change noticeably.

- **Empty space skipping** performs no operations on regions of a scene that do not noticeably contribute to the pixel's final color.

- **Accelerated (Semi) Iso Surfacing** means ray tracing the implicitly given (and therefore not explicitly stored) iso surface by additionally skipping regions of no interest. The regions of interest depend on the iso value s_{iso} and the iso surfacing mode activated. For deciding if a region is of interest, the region has to be equipped with an interval $[s_{min}, s_{max}]$ that indicates the minimal/maximal scalar values given inside the region. A region is only of interest if $s_{min} \leqq s_{iso} \leqq s_{max}$ (iso surfacing) or $s_{max} \geqq s_{iso}$ / $s_{min} \leqq s_{iso}$ (semi iso surfacing).

  Note that accelerated iso surfacing allows to change the iso surface on the fly, i.e., neither the iso surface nor the acceleration structure need explicitly to be re-calculated and stored. The implicitly given iso surface is directly displayed in the next rendering pass.

1.5 Hierarchical Acceleration Structures

Choosing a good acceleration structure to a given scene and visualization purpose is cruicial. Acceleration structures are therefore a major research topic for interactive ray casting/ray tracing, and an own section is devoted here for shortly introducing different acceleration techniques. A comparison of various hierarchical acceleration structures may be found in Havran's PhD thesis (15).

1.5.1 Spatial Partitioning Hierarchies

Spatial partitioning hierarchies partition space, where each spatial region is referenced only once. They have therefore multiple references to primitives that overlap different spatial regions.

1.5.1.1 bsp-tree

Binary space partitioning trees recursively subdivide the space by some arbitrary positioned and oriented split plane (10).

Some approaches of using bsp-trees and restricted bsp-trees (bsp-trees that have pre-defined possible split plane orientations) for interactive ray tracing are (19), (5; 20).

1.5.1.2 *k*d-tree

k dimensional trees are bsp-trees with the restriction that their split planes have to be oriented parallel to one of the main axes (4). As indicated in (15), for most scenes *k*d-trees built using a surface area heuristic (13; 25) allow probably the best render times.

Simple constructions are fairly fast and allow interactive rebuilds for small scenes (17; 40). More advanced constructions are further optimized for visualization speed: The bounding boxes of primitives straddling the *k*d-tree's split planes are re-adjusted to tightly fit the primitive's subparts belonging to the children nodes of the split node (51). Advanced constructions are typically for large scenes to slow for interactive rebuilds.

1.5.1.3 (hierarchical) grid

Simple grids are not well suited for the "teapot in the stadium problem", where large portions of the primitives reside in very few cells of the grid. Hierarchical grids subdivide gridcells containing too much primitives into new grids, what resolves that problem. One of the first interactive ray casting systems working on a supercomputer did use hierarchical grids as acceleration structure (37). More recently has a coherent grid traversal technique been introduced (52).

1.5.2 Object Partitioning Hierarchies

Object partitioning hierarchies partition the set of primitives (objects), where each primitive is referenced only once. They have therefore multiple references to spatial regions that are overlapped by different primitives.

1.5.2.1 BVH

The bounding volume hierarchy is a tree of bounding volumes, where the bounding volume at a given node encloses the bounding volumes of its children (7). For interactive ray tracing are exclusively axis aligned bounding boxes as bounding volumes (48) used. Its construction is quite fast and allows interactive rebuilds for small to medium sized scenes (18; 47).

1.5.2.2 s*k*d-tree

Spatial k dimensional trees are similar to kd-trees, with the exception that they allow to partition spatially overlapping objects, where not one but two parallel split planes partition the objects within a node (36). They are therefore a mixture of kd-trees and BVHs, where they have a fast traversal similar to that of a kd-tree, but they are an object partition hierarchy just like BVHs. Some of the current state of the art interactive ray tracers use skd-trees (16; 45; 59).

1.6 Complexity Orders for *k*d-trees and s*k*d-trees

This section is devoted to the orders of the construction-, update- and visualization-time of kd-trees built above rectilinear grids and skd-trees built above unstructured primitives.

In the complexity analysis the focus is on the complexity of construction-update- and render-time with respect to the amount of primitives n. The render time t does of course not only depend on n but also on the amount of pixels p and processor cores c used. But this dependency is basically linear such that the focus is only on $t(n) \approx t(n, p, c)c/p$.

1.6.1 Construction and Update Time

Tree construction is done only once when loading the data set. It is for kd-trees built above a rectilinear grid of order $\mathcal{O}(n)$ and for skd-trees of order $\mathcal{O}(n\ lg(n))$. If additionally some surface area heuristic is implemented the orders increase (for kd-trees e.g., to $\mathcal{O}(n\ lg(n))$). Tree construction behaves therefore for large

data sets nearly linear and is through its recursive nature efficiently parallelizable upon multicore processors. If a scene is deformed, the underlying s*k*d-tree does not need to be reconstructed, but may be updated which is of linear order $\mathcal{O}(n)$.

1.6.2 Render Time: Worst Case Scenarios

In worst case scenarios, brute force visualization of n opaque objects takes $\mathcal{O}(n)$ operations. If the objects are semi-transparent, and some non-commutative blend function shall be applied correctly, the objects have to be additionally sorted, which increases the amount of operations to $\mathcal{O}(n\ lg(n))$. This holds for both image order and object order techniques.

1.6.3 Render Time: Real Life Examples

But this holds only for artificial constructed worst case scenarios. For reasonable scenes the order for opaque ray tracing by using an underlying (s)*k*d-tree is rather of order $\mathcal{O}(lg(n))$.

1.7 Comparison to Hybrid and Object Order Techniques

Hybrid and object order techniques both iterate at least once over all primitives inside the view frustum and are therefore at least of linear order $\mathcal{O}(n)$ or even worse of order $\mathcal{O}(n\ lg(n))$. This stands in contrast to ray casting accelerated through the usage of (s)*k*d-trees which is of sublinear order $\mathcal{O}(lg(n))$ (opaque or accelerated (semi) iso surface rendering) or $\mathcal{O}(n^{1/3})$ (volume rendering). For static scenes ray casting does outperform object order and hybrid techniques at some given data size.

Chapter 2

Hybrid Visualization: Interactive CPU Ray Casting Combined with GPU Rasterization

2.1 Abstract

A new hybrid visualization technique is introduced here. It is controlled over an OpenGL friendly interface and allows the simple and fully interactive integration of different ray casters into already existing OpenGL frameworks or vice versa it supports to render primitives drawn by the graphics card into ray cast scenes. The interface also allows to run the CPU-based ray casters and the GPU-based rasterization in parallel and furthermore supports shadows.

The technique allows to combine the advantages of today's ray casters and graphics cards. Graphics cards provide many functionalities, and are optimal for the interactive visualization of medium sized dynamic scenes consisting of triangles, lines and texture mapped objects. Ray casters are optimal for the interactive visualization of very large static scenes consisting of primitive types with fast ray-primitive intersection tests.

2.2 Problem

CPU-based ray casting surely has certain advantages compared to GPU-based rasterization, but this holds also vice versa.

Restricting oneself to one of both visualization techniques means loosing the possibilities offered by the discarded technique and is therefore not a satisfying choice. But this is commonly done. Visualization systems are either tied to the graphics card or to a ray caster. Hybrid visualization systems do actually exist, but they all pursue the goal of either trying to improve the visualization quality of a rasterized image, or trying to speed up the ray casting/ray tracing process. The task of interactively drawing rasterized and ray cast primitives together into one image has up to now not been tackled.

2.3 State of the Art

The general idea of combining rasterization with ray tracing is not new. But hybrid visualization techniques are commonly used to either improve the visualization quality of a rasterized image, or to speed up the ray casting/ray tracing process.

Stamminger et al. and Beister et. al (3; 43) generate high quality images by combining ray tracing with rasterization. The rasterization algorithm is used to replace the casting of primary rays. Secondary rays are then cast by the ray tracer to generate the final image. This approach is capable of creating pseudo photo realistic images at fairly interactive frame rates but works for at most medium sized scenes, since the entire scene is also stored on the graphics card. Balciunas et al. (2) improved ray casting performance of height fields by using the graphics card to draw rough bounding volumes of the height field into the graphics card's depth buffer. The depth buffer is then send to the ray caster which uses it to initialize rays close to primitives. This minimizes the traversal steps through an acceleration structure.

The approach described in this chapter differs from those above, as it concentrates on combining the functionalities and the visualization powers of modern ray casters and graphics cards for scientific visualization. Modern ray casters are

- when aiming at the interactive visualization of large static scenes - already very powerful, such that they do not need the GPU's assistance.

Interactive ray casting has been first achieved on supercomputers (Muuss et al. (33), Parker et al. (39)). Interactive ray tracing on desktop machines for large static scenes has been first achieved by Wald et al. (46). Wald et al. combined SIMD operations with fast traversal of highly optimized *k*d-trees build in a pre-processing step by using a surface area heuristic (13; 25). Interactive ray tracing of small up to medium sized *dynamic* scenes has recently become a more investigated research topic. Some acceleration structure is for each frame interactively constructed (e.g., (52), (44)) or updated (e.g., (48)). But those techniques suffer the same problem as the graphics cards do. Building or updating the acceleration structure is at least of order $O(m)$, where m is the amount of moved primitive groups. Ray casters for completely dynamic scenes are not as fast as graphics cards, since they do not have direct hardware support. When aiming at the non-photo-realistic interactive visualization of dynamic scenes, graphics cards are (still) the best choice.

2.4 Results

Ray casters are perfectly suited to visualizate large static data sets. But render kernels based purely on ray casting have all the same basic problems. It is tedious to:

- integrate the ray casters into already existing OpenGL frameworks (e.g., replacing an on marching cubes based iso surface rasterizer by an accelerated iso surface ray caster)
- display useful metadata additionally to/into the data set (e.g., the scene's bounding box, a scale, text)
- display additional dynamic objects or lines into the scene (e.g., flowing particles with their trajectories, streamlines)

The solution to all those problems is to use a hybrid viewer that has an OpenGL friendly interface which allows to display the metadata and the non-ray cast objects using the graphics card.

When searching for experience reports or publications that tackle this problem, one realizes that hybridviewers do exist, but they all pursue a different goal, i.e., they either try to improve the visualization quality of a rasterized image, or try to speed up the ray casting/ray tracing process. But the problem described here - drawing rasterized and ray cast primitives interactively together into one image - remains unanswered.

A new hybrid viewer is hence introduced in here, where further requirements are pursued:

- The hybrid viewer's setup time has to allow interactive frame rates.
- The CPU and the GPU shall run in parallel, i.e., the overall render time shall not be the sum, but roughly the maximum of both render times (ray casting and rasterization).
- The interface shall - just like the ray casters - support SIMD operations and multiple CPU cores.
- Multiple ray caster kernels shall be supported, i.e., the viewer may merge the OpenGL image together with images of multiple ray casters.
- As the focus is on scientific visualization, advanced illumination effects such as reflections and refractions are not of high interest, but shadows may be of interest, such that they shall be supported.

The hybrid viewer described in this chapter does fulfill all those goals and is the basis and the interface to the ray casters given in this thesis.

Even though the contents of this chapter may not be strikingly novel or original, it may be that others in the visualization- (and especially in the ray tracing-) community may have encountered similar problems as desribed here, such that this chapter shows quite detailed how the hybrid viewer is built, where some examples of application are also given.

2.5 The Hybrid Visualization Technique

This section gives very precise descriptions to the hybrid visualization technique. Many of the information given here may be seen as basic knowledge, such that experienced readers may briefly read over this section. Nevertheless detailed information is provided here not only for the sake of completeness, but also for non-OpenGL or non-ray tracing experts.

The hybrid visualization technique uses the CPU for ray casting and the GPU for rasterization. The resulting images of both visualization passes are merged into the final image by using the color- and the depth-buffer.

2.5.1 Fast Ray Generation

Ray generation is affected by the camera's projection type. The two most commonly used projection types are parallel projection and perspective projection, where the later is chosen as explanation example.

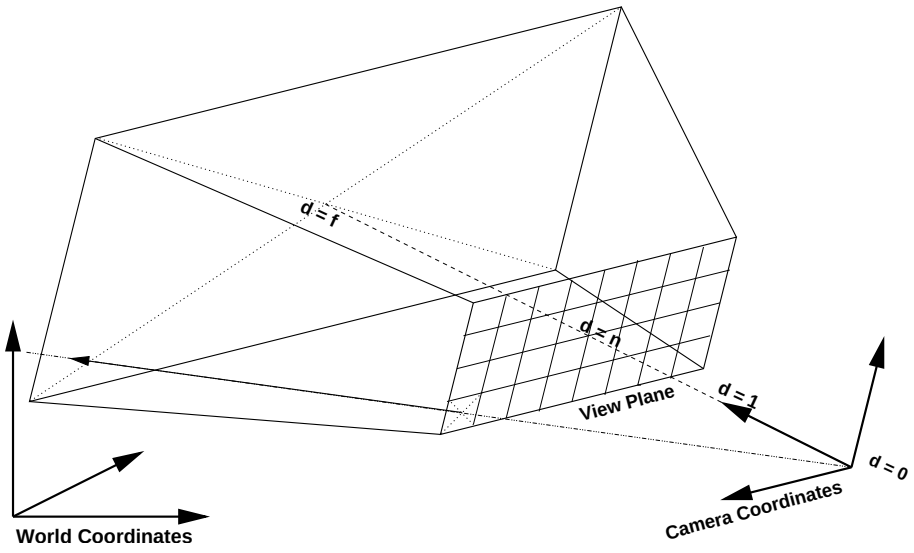

Figure 2.1: Camera coordinates, viewport and view frustum relative to the world coordinates. The ray belonging to the lower left pixel of the viewport $R[0][0]$ is shown in the view frustum.

The following information is passed towards the interface, which then generates rays and passes them over to the ray casters.

1. Modelview matrix M: contains the information of the camera's position and view direction

2. Projection matrix P: contains the information of the camera's view frustum and projection type (parallel or perspective)

3. Viewport: width w and height h of the view plane's pixel resolution

These parameters are obtained from OpenGL by using the function `glGet()` (34), where the modelview matrix M and the projection matrix P are returned as arrays of 16 values with row-wise (and not column-wise) storage order. This data is used by the interface for reconstructing the information needed for ray generation.

The camera position and view direction is retrieved out of the modelview matrix by inverting it. OpenGL uses a right handed coordinate system, while the ray casters of this thesis use a left handed coordinate system. The first three entries of the matrix' third row are therefore multiplied by -1. In the resulting matrix the first three entries of the first three rows are the direction of the camera's x, y, and z-axis (cam_x, cam_y, cam_z) in world coordinates. The first three entries of the last row is the camera's position (and therefore the origin of all rays R_O) in world coordinates.

The direction $R_D[i][j]$ of the ray belonging to the pixel with viewport-coordinates $i \times j$ is retrieved out of the projection matrix. The last entry of the projection matrix $P[15]$ specifies if the camera mode is either perspective ($P[15] = 0$) or parallel ($P[15] = 1$) projection. OpenGL generates the following matrix for perspective projection.

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Where n and f are the distances of the frustum near and far plane to the camera. Their values are retrieved from the projection matrix by the equations

$$n = P[14]/(P[10] - 1) \qquad f = P[14]/(P[10] + 1)$$

The by n divided left l', right r', bottom b' and top t' planes of the view frustum are extracted by the formulas

$$l' = (P[8] - 1)/P[0] \qquad r' = (P[8] + 1)/P[0]$$
$$b' = (P[9] - 1)/P[5] \qquad t' = (P[9] + 1)/P[5]$$

The changes in ray direction $\Delta_w R_D$ and $\Delta_h R_D$ when moving from one pixel to one of its neighbor pixels are given by the equations

$$\Delta_w R_D = ((r' - l')/w)cam_x$$
$$\Delta_h R_D = ((t' - b')/h)cam_y$$

The ray direction of the lower left pixel $R_D[0][0]$ is

$$R_D[0][0] = cam_z + l'cam_x + b'cam_z + 0.5(\Delta_w R_D + \Delta_h R_D)$$

The ray direction $R_D[i][j]$ is then evaluated by

$$R_D[i][j] = R_D[0][0] + i\Delta_w R_D + j\Delta_h R_D$$

The entire process is not repeated for each single pixel, since this is too expensive when aiming at interactive visualization. Instead the values l', r', b', t', n and f are only evaluated at initialization and after changes of the projection matrix. R_O, $\Delta_w R_D$, $\Delta_h R_D$ and $R_D[0][0]$ are for each frame computed once and stored. Each ray direction is then directly computed by the last given formula.

2.5.2 Fast Depth Buffer Conversion

The OpenGL depth buffer is not linear in world space. Its values are in the range [0, 1], where 0 stands for points on the near plane with linear distance n and 1 stands for points on the far plane with linear distance f. Values in (0, 1) cannot be transformed to a linear depth buffer by a linear function, since the OpenGL depth buffer's precision decreases when moving from the view frustum's near to its far plane. For fast buffer conversion the two help variables b_1 and b_2 are evaluated at initialization and after changes of the projection matrix.

$$b_1 = fn/(n - f) \qquad b_2 = f/(f - n)$$

The buffer conversions between the OpenGL and the linear buffer and vice versa are then performed with the two formulas.

$$d_{linear} = b_1/(d_{OpenGL} - b_2) \qquad d_{OpenGL} = (b_1/d_{linear}) + b_2$$

2.5.3 Image Merging

Image merging can either be executed explicitly or implicitly. Explicit merging waits for both result images of each visualization process and merges them to-

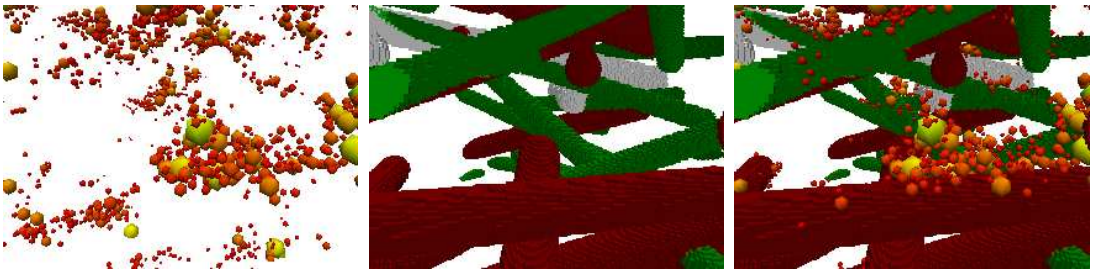

Figure 2.2: Rasterized, ray cast, and merged image

gether into the final result image. Implicit merging waits for one result image of one visualization process and sends it to the other visualization process which uses it as initial color- and depth-buffer. The image generated by the second visualization process is then the final image.

At the first glance explicit merging seems to be a better choice, since it allows both visualization processes to run in parallel, in contrast to the sequential visualization order needed for implicit merging. But implicit merging with rasterization as initial visualization technique is choosen here, since it offers other major benefits.

1. The second visualization process is not tight to the visualization of opaque objects, but may also be used to visualize semi-transparent objects

2. When using rasterization before ray casting the clearing of the buffers is performed with direct hardware support on the graphics card

3. When using rasterization before ray casting it may speed up the ray casting process

The fact that the ray casting process may be sped up is explained as follows. When using explicit merging, the ray caster shoots rays the entire distance from the frustum's near plane to its far plane. When using implicit merging with rasterization as initial visualization technique, the ray caster only shoots rays from the frustum near plane to the nearest object visualized by the rasterization algorithm. Rays may therefore terminate before they hit an object visualized by the ray caster. Hence, unnecessary traversal steps and ray-primitive intersection

tests are saved through this early ray termination (see Figure 2.3). This has a positive effect on the overall ray casting time.

Double buffering is used for not loosing the ability to use CPU and GPU in parallel. The graphics card draws the initial image. The resulting color- and depth-buffer are sent to the ray caster. The ray caster starts to draw into this image. Parallel to this, the graphics card starts to visualize the next image.

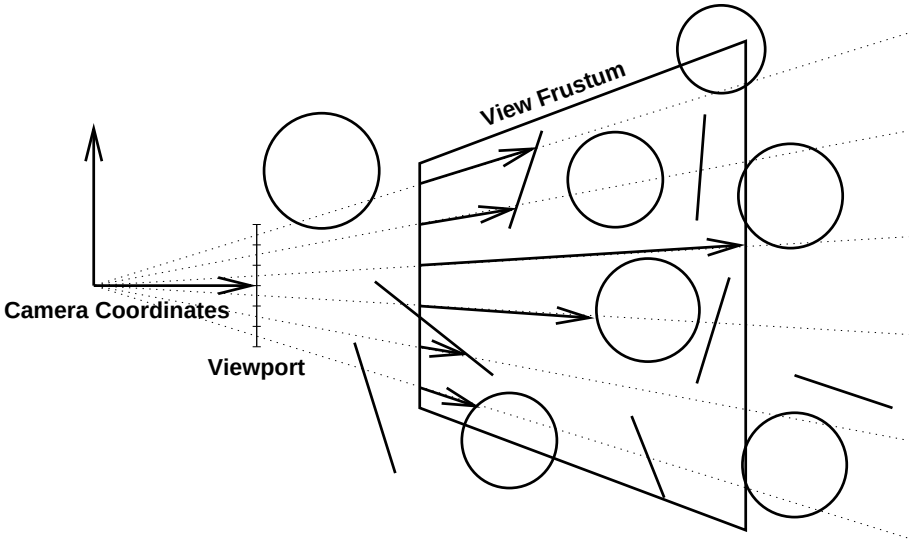

Figure 2.3: 2D example of early ray termination, when using implicit image merging. Some rays that would have hit objects visualized by the ray caster (circles) are early terminated by objects initially drawn by the graphics card (lines)

2.5.4 Viewport Minimization: The Bounding Viewport

All the above mentioned operations are performed on each pixel in the viewport. But in scientific visualization it is common that objects are observed in such a manner that they do not fill the entire viewport (e.g., one may examine an object while "holding" it in front to view it). For this case many pixel operations are not needed, as the corresponding rays created and merged do not even touch the ray cast scene's bounding box. Therefore viewport minimization is also applied, where only pixels are considered that may intersect the scene.

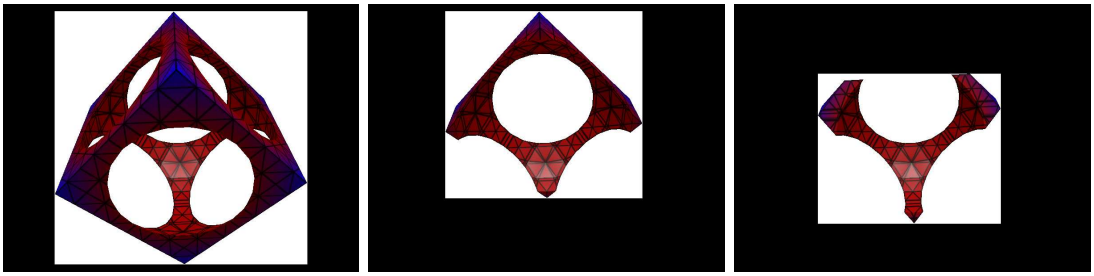

Figure 2.4: Bounding viewports to a ray cast data set (here iso surface and regions containing values smaller than the iso value of a tetrahedral mesh), where the data set is shown full, clipped along one of its main axes, and clipped against the view frustum's near plane.

The bounding box of the scene to be ray cast is clipped against the view frustum, the resulting polyhedron is projected onto the viewport. The minimal rectangle of pixels in the viewport that contains that entire projection is the bounding viewport. Rays belonging to pixels outside the bounding viewport cannot intersect the ray cast scene, such that the above mentioned operations are only performed inside the bounding viewport. Note that this also works for cases where the bounding box of the ray cast scene does not coincide with the bounding box of the rasterized scene.

It has to be kept in mind that OpenGL uses a non-linear depth buffer, such that near/far plane clipping cannot be performed in homogeneous coordinates. Therefore the bounding box is first transformed into modelview space where the near/far plane clipping is performed (since the depth buffer is still linear there). The resulting polyhedron is then transformed into homogeneous coordinates where the top/bottom and left/right plane clipping is performed.

2.5.5 The OpenGL-Friendly Interface

Many applications require that OpenGL API routines are called out of the main thread. Parallel rasterization and ray casting is therefore only possible, if all interface- and ray caster-calls are executed in separate threads in the background, while the OpenGL calls are executed out of the main thread. The interface and the ray casters are therefore written thread safe.

For integration of a ray caster into an already existing OpenGL viewer, the function `hybridDisplay()` is introduced (Algorithm 1). It will be called instead of the `display()` function and takes care of the buffer handling and the synchronization of the ray casting- and the rasterization-process. Internally it calls the `display()` function when needed. Projection matrix and viewport are only updated at initialization and after viewport resizes and are therefore neither updated nor transferred in this function.

The `startDisplay()` function of the interface class activates a separate thread and returns immediately, while the separate thread continues in the background by starting a common parallelization procedure.

The separate thread uses the amount of processor cores c available on the machine. It divides the image into tiles whose amount is a multiple of c, initializes a tile stack, and activates c sub-threads. Each sub-thread pops one tile at a time from the tile stack, renders it and continues with the next tile on top of the stack until the stack is empty.

The sub-threads perform the following operations per pixel in their current image tile. The depth value is converted, a corresponding ray with the appropriate start and end time is created and passed over to the ray caster assigned to the interface. The ray caster returns a color with an additional alpha channel. The final pixel color results by using this alpha channel to merge the initial pixel color given by the rasterization process with the pixel color returned by the ray caster.

The work load is well balanced between the cores, since the image rendering is subdivided into a work queue, where each core grabs work queue elements on demand. The function `endDisplay()` waits until all sub-threads are done. The interface's color buffer holds then the final image.

Note that in this setup the ray casters are not specified. Arbitrary ray casters may be assigned to the interface to which the rays are then passed over.

2.6 Shadow Visualization

Shadows may be an important element for scientific visualization. They provide significant visual clues, aiding the user to understand spatial relationships in

Algorithm 1 Hybrid Display Function

```
void hybridDisplay()
{
  interface->setModelviewMatrix(getModelviewMatrix());
  glLightfv(GL_LIGHT0, GL_POSITION, interface->getLightPosition());
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  if(rasterization)
  {
    display();
    readColorBuffer(interface->getColorBuffer(),
                    interface->getBoundingViewport());
    readDepthBuffer(interface->getDepthBuffer(),
                    interface->getBoundingViewport());
  }
  if(rayCasting)
    if(doubleBuffering)
    {
      interface->endDisplay();
      interface->swapBuffers();
      interface->startDisplay();
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    }
    else
    {
      interface->startDisplay();
      interface->endDisplay();
    }
  if(rayCasting)
    writeColorBuffer(interface->getColorBuffer(),
                     interface->getBoundingViewport());
  glSwap();
}
```

a scene (see Figure 2.5). This section's focus is on efficiently adding shadow visualization to the hybrid visualization technique described above in 2.5.

The literature on shadow generation is vast and exceeds the scope of this

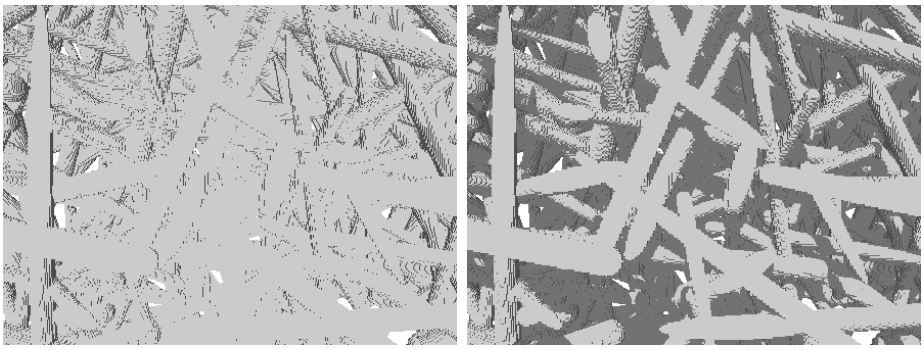

Figure 2.5: Gray colored fibres visualized unshadowed and shadowed. Clearly shadow visualization helps to understand spatial relationships.

section. For an old but nice survey on shadow generation see (58). The focus here is on shadow depth maps and the casting of shadow rays.

Shadows are simulated in ray casting by simply casting secondary rays towards the light source. A point is in shadow if its shadow ray hits some object before it reaches the light source.

One way to simulate shadows with rasterization are shadow depth maps. The scene is first rasterized from the light's point of view. From this rendering, the depth buffer (the shadow depth map) is extracted and saved. Since only the depth information is relevant, it is common to disable all texture calculations and not to update the color buffer for this rendering. The scene is then rendered twice out of the camera's perspective, once totally shadowed and once totally unshadowed. To each pixel is a depth value given which defines a point in world coordinates. This point is transformed into light coordinates. The point is in shadow, if its transformed depth value is greater than the depth value indicated by the corresponding point on the shadow depth map. Depending on this look up the corresponding pixel is colored shadowed or unshadowed.

2.6.1 Small and Precise Shadow Depth Maps

The shadow depth map has to be small and precise. It needs to be small (i.e., consist out of as few pixels as possible), since it is transferred from the graphics card to main memory. It needs to be precise (i.e., consist of enough pixels and

of precise depth values), because the shadow shall not reveal that it has been discretized to a shadow map and no artifacts shall appear when comparing a point's converted depth value with the corresponding shadow depth map value.

The light's position does strongly affect the precision of the shadow depth map. Fortunately, the light's position for scientific visualization is typically not of special interest. The light has to be placed only at some point such that the structure of the scene/object observed is clearly revealed. The light's position is therefore clamped to be always somewhere directly above the camera, where the user may still choose a light angle. This leads to appropriate illumination of the scene, as the light subjectively comes always from somewhere above the observer.

This choice has also a further major advantage. It allows to construct a shadow depth map that is row-wise perfectly precise (i.e., pixels in the i-th column of the viewport can only be shadowed by pixels in the i-th column of the shadow depth map).

The light's coordinate system is chosen to be identical to the camera's coordinate system, except that it has been translated along the camera's z-axis. The light's projection matrix is also chosen to be identical to the camera's projection matrix, only the top/bottom t/b values are modified (Figure 2.6).

Similar to the bounding viewport, a bounding shadow map is evaluated. The evaluation of a precise shadow map needs only minor additional steps as for the bounding viewport.

The bounding box of the entire scene is clipped against the view frustum. The resulting polyhedron's minimal/maximal z-values in light coordinates are first used to update the light's projection matrix near/far values (this increases the shadow map's precision). The polyhedron is then projected onto the light view plane. The light viewport is set to the same size as the viewport. The minimal rectangle of pixels in the light viewport that does contain that entire projection is the bounding shadow map.

Using this technique and $f \leq 25n$, the constructed shadow map does not display major artifacts.

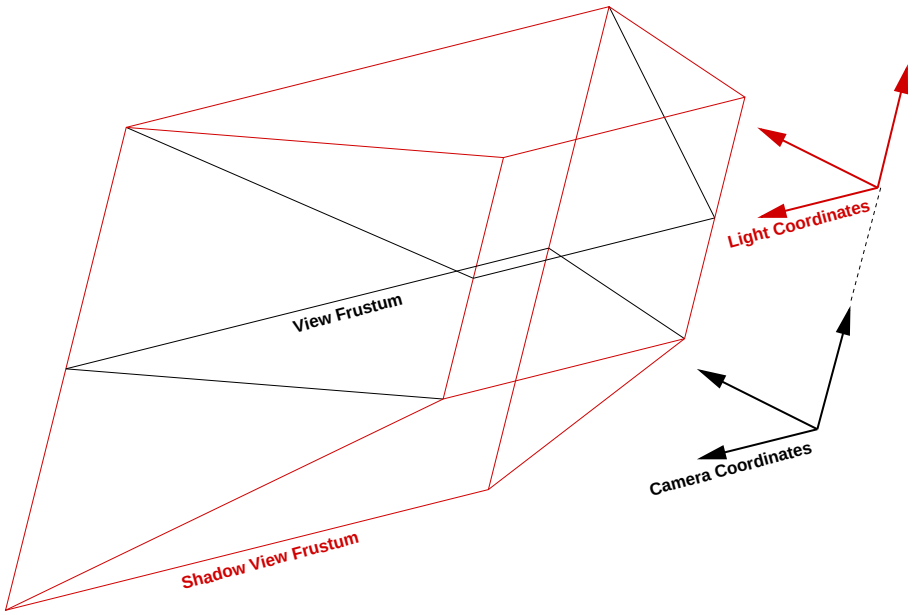

Figure 2.6: View frustum and shadow view frustum

2.6.2 Setting the Light's Angle

In the setup described here, the light is always placed somewhere directly above the camera. The user may define a light angle, where the angle does not rotate but only translate the light's coordinate system. The translation's length is defined by the light angle and the distance of the scene to the observer.

The entire scene's bounding box is clipped against the view frustum. The midpoint of the resulting polyhedron's minimal and maximal z-value in modelview space is then taken as the distance of the scene's center to the observer. This techniques ensures that the scene's distance to the observer is always positive, i.e., the light is always above the observer.

2.6.3 Applying the Shadow Depth Map

As the light is always somewhere above the observer the shadow depth map changes after every observer (camera) move. The hybrid visualization interface evaluates therefore after every camera move the new light position and direction (light modelview matrix), a corresponding minimized shadow depth map, and

a shadow matrix, which is the product out of the light modelview-, the light projection-, the 1/2-scaling-, and the 1-translation-matrix. This shadow matrix is then used for determining if a point in world coordinates is shadowed by the shadow depth map.

The interface multiplies a point's coordinates with the shadow matrix and performs a perspective division on the result point. The new point is now represented in scaled and biased homogeneous light coordinates, where the x, y, and z-values of eventually shadowed points are in the range $[0, 1]$. The x and y value of this point are multiplied with the viewport's width and height, respectively, and rounded to the integers i and j. The original point is in shadow, if the $(i \times j)$-th shadow pixel belongs to the bounding shadow map and if the corresponding depth value is smaller then the z value of the transformed point.

This technique allows to quickly answer if some point in world coordinates is shadowed by the shadow depth map. If this is the case, no shadow rays need to be cast. This saving of casting shadow rays is comparable to early ray termination. But the speed up achieved here is higher as for early ray termination, as in this case shadow rays need neither to be cast nor even to be generated.

2.6.4 Combining Shadow Depth Mapping and Shadow Ray Casting

The hybrid shadow renderer tries to minimize the casting of shadow rays since this is far more expensive than shadow map look ups. Shadows are therefore visualized as follows.

The light modelview- and light projection-matrix are sent to the graphics card, which then first draws the shadow depth map, and then renders the scene twice from the camera point of view, once in, and once not in shadow. The resulting shadow color buffer and the shadow depth map are sent (besides the color- and the depth-buffer) to the hybrid visualization interface. A transformation of the shadow depth buffer to a linear depth buffer is not needed.

The interface generates now for each pixel a ray and passes it over to the ray casters. If the ray hits some objects the interface checks if this hit point is already shadowed by the shadow depth map. If so, no secondary ray needs to be shot

and the corresponding pixel is colored in shadow. Otherwise the conservative procedure of generating and casting a shadow ray is performed.

If no ray cast object is hit, the interface first checks, if the pixel's corresponding depth value is smaller than the far value f. If this is not the case, the pixel is not overlapped by any object, and no secondary ray is cast. Otherwise the object's hit point (which has been drawn by the graphics card) has to be checked if it is in shadow or not. First the shadow depth map is checked. Only if the result is *not* "in shadow", a shadow ray is cast. If the point is not in shadow, the final color is equal to the color indicated by the color buffer, otherwise it is equal to the color indicated by the shadow color buffer.

2.6.5 Usage of SIMD Operations

Interactive ray casters typically use SIMD operations in order to traverse acceleration structures and to intersect primitives with bundles of multiple rays in a packet. The interface of the hybrid visualization technique also uses SIMD operations, where ray generation, depth buffer conversion, image merging and even shadow casting are all performed for ray bundles. The interface passes SIMD-friendly packets of rays over to the ray casters, and merges the resulting image with the image generated by the graphics card.

Only shadow depth map look ups are not performed by SIMD operations, since adjacent pixels on the image plane may have non-adjacent shadow pixels in the shadow depth map. Shadow rays on the other hand are also cast in ray bundles. Rays that are already shadowed by the shadow depth map are deactivated through masking operations, such that only shadow rays that are actually needed are cast.

2.7 Computational results

A dual-dualcore 2,3 GHz Intel Xeon 5148 (64 bit) equipped with a NVIDIA GeForce 7800 PCIe 16x graphics card has been used as testing environment. The viewport size was set to 800×600 pixels.

2.7.1 Pure Setup Time

The pure setup time is important since it gives the upper bound of the framerates that can be achieved using the hybrid visualization. It has been evaluated by visualizing the 6 opaque boundary squares of the unit cube via rasterization. All four CPU-cores have been activated by the means of starting four sub-threads each handling ray generation, shadow map look ups and image merging.

OpenGL	+Transfer	+Ray Gen.	+Merg.	+Shadow
0.3 (3333)	5 (200)	8 (125)	11 (91)	19 (52)

Table 2.1: Pure setup times in milliseconds (frames per second) for the rasterized unit cube.

The columns in Table 2.1 stand for:

1. Pure OpenGL visualization, i.e., clearing the buffers, drawing the cube and displaying the result.

2. Additional transfer of the color- and the depth-buffer to the interface, and returning the unmodified color buffer back to the graphics card.

3. Additional depth buffer conversion and ray generation.

4. Additional merging process of the ray cast and the rasterized image (i.e., full setup time for hybrid visualization without shadows).

5. Setup time or full hybrid visualization with shadows i.e., clearing color- and depth-buffer, drawing the cube and sending the color and depth buffer. Clearing the buffers a second time, drawing the cube in shadow and sending the shadow color buffer. Clearing the depth buffer a third time for drawing and sending the shadow depth buffer. The interface then generates rays, passes them over to an inactive ray caster, performs then for each ray a shadow map look up (since the inactive ray caster always returns "no hit"), passes rays of un-shadowed points over to the inactive ray caster (the points stay un-shadowed), draws the scene according to the shadow map look ups and sends the final color buffer back to the graphics card which displays it.

Even though no real ray casting process itself is started (the inactive ray caster directly returns the result "no hit"), shadow visualization via shadow mapping is already performed in the setup described above.

The results show that the setup time is fully interactive and achieves real time frame rates even with shadowing activated.

2.7.2 Parallel Visualization

The parallel usage of CPU and GPU is independent of the scene, of the graphics card and of the ray caster used. It suffices to use a single example for verifying that both visualization processes are running in parallel.

The visualization of oil filtration has therefore been choosen, as it gave the final impulse for this work. The oil flow through the filter has been calculated on a cartesian grid. The computational results are best verified by visualizing the particles dynamically flowing through, and being filtered inside this cartesian grid.

The filtration process chosen consists of an oil filter given as an 800^3 cartesian grid with more than 347 million voxels representing the filter's fibers. 96000 particles are filtered out of the oil (Figure 2.7). This scene is challenging for pure rasterization or pure ray casting respectively.

Even the fully optimized boundary representation of the fibers still consists of more than four millions of rectangles (or twice as many triangles) and is therefore challenging for the interactive visualization on today's graphics card. A ray caster is best suited for this scene, since the ray-voxel intersection tests are very fast, and the filter itself is static. Visualizing the dynamically moving particles interactively is non-trivial for ray casting, but efficiently realized on a graphics card. The particles are in this setup represented as icosahedras (i.e., 20 triangles per particle) resulting in an overall of 1.92 million triangles.

The ray caster used here fully supports SIMD-extensions and uses an implicit bitmask *k*d-tree (see Chapter 3). The tree's storage requirement is only four bit per grid element. Tree construction uses all available CPU-cores and takes for this particular scene 4.2 seconds. The resulting tree needs only 256 megabyte in main memory.

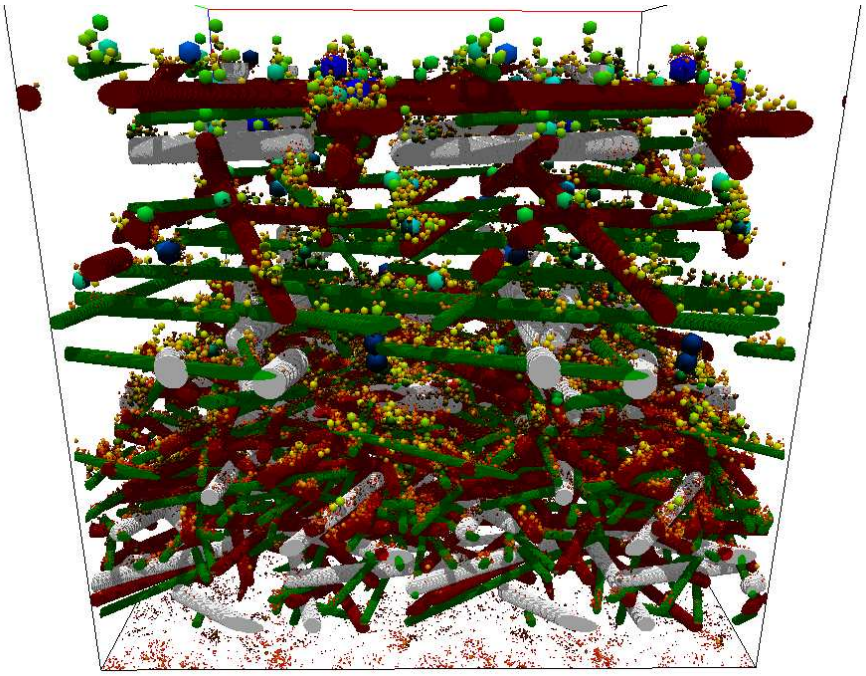

Figure 2.7: 96000 particles in an 800^3 oil filter, visualized with 3.26 (shadows) and 8.8 (no shadows) frames per second

illumination	rasterizing	ray casting	hybrid
local	98 (10.2)	91 (11.0)	113 (8.80)
shadow	305 (3.28)	171 (5.81)	306 (3.26)

Table 2.2: Visualization times in milliseconds (frames per second) for the filtration process. First: Only the dynamic parts are visualized by the graphics card. Second: Only the static parts are visualized by the ray caster. Third: Parallel visualization of the full scene.

The results in Table 2.2 clearly show that CPU and GPU are running in parallel. The rendering time for the full scene without shadows already shows that visualization is partially executed in parallel, since the overall rendering time is faster than the sum of both single visualization times. The effect of parallel rendering is fully recognizable when displaying the entire scene with shadows. This takes almost exactly the same time as only rendering the particles via rasterization and shadow mapping. The ray- and shadow-casting process is fully

hidden behind the rasterization time when activating both visualizations in parallel. Pure rasterization combined with shadow mapping takes roughly triple the time needed than pure rasterization without shadows. This is due to execution of the three rendering passes needed for shadow mapping.

2.8 Applications and Experiences

The hybrid visualization technique has originally been used to map scalar fields upon rasterized objects and to display useful metadata (Figures 2.8, 2.9).

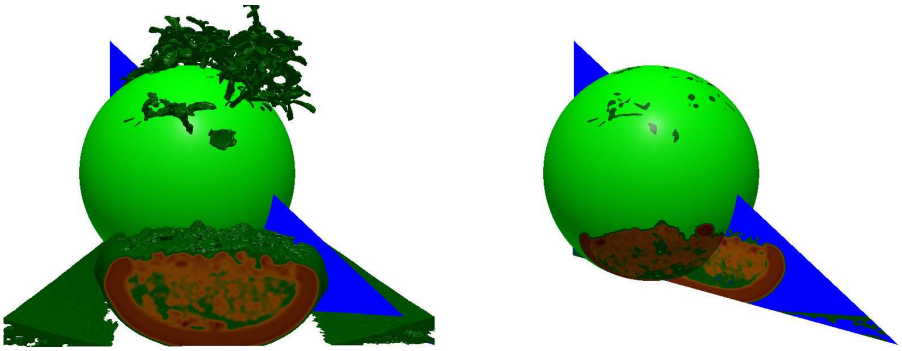

Figure 2.8: Result mapping: A rasterized triangle and a ray cast sphere are shown together with the ray cast bonsai scalar field. The later is shown once by semi-isosurfacing and once mapped onto the other objects.

Different ray casting kernels of this thesis have also been shown to members of the ITWM's department for flow and material simulation (SMS). They - experts in microstucture simulation and hydrodynamics - showed strong interest in this work, such that two ray casting kernels had been implemented for them.

One is a structure viewer especially designed to visualize large microstructures generated by the SMS' software tool GeoDict. GeoDict actually already contained an own structure viewer based on rasterization using OpenGL. But the SMS favors the new ray casting based solution, as it allows to interactively slice through the data set without displaying artifacts, it has faster setup times, and it achieves even for large data sets interactive framerates.

The other ray casting kernel allows new features that prior have not been implemented in GeoDict. It is an advanced accelerated iso surface viewer designed

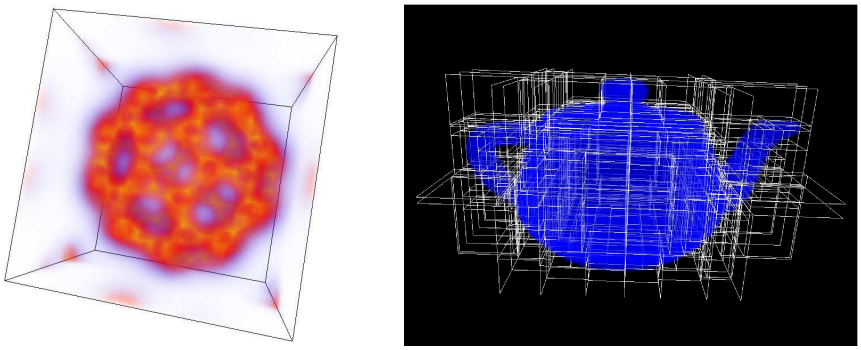

Figure 2.9: Left: A volume ray cast tetrahedral mesh with its bounding box. Right: The rectilinear grid representation of the Utah teapot volume ray traced, where the planes to an underlying *k*d-tree build using a SAH are displayed with lines drawn by OpenGL. The images show that the hybrid visualization technique allows the merging of semi transparent ray casters with opaque OpenGL scenes.

for displaying not only iso surfaces, but also regions greater/smaller than the iso value upon which additional scalar values may be displayed using coloring and iso lines.

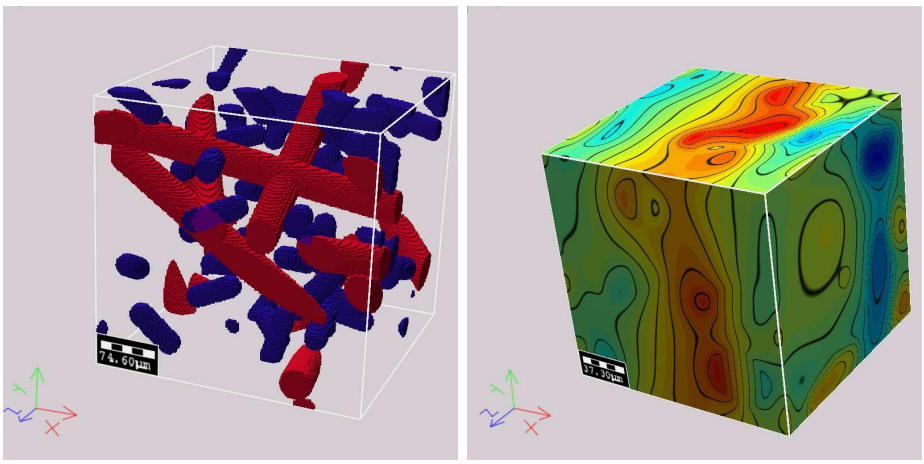

Figure 2.10: Two ray casting kernels integrated into an already existing OpenGL framework. The meta data which is the bounding box, the coordinate axes and the texture mapped scale are displayed by OpenGL, while the data is displayed by the ray casting kernels.

Unfortunately, the SMS' visualization group is not very familiar in using ray

casters, but they are well experienced in using OpenGL as they also built OpenGL based render kernels for visualizing different simulation results of GeoDict. Therefore the OpenGL friendly interface has been refined to the state as described in this chapter. A small OpenGL example program has then been supplied to SMS. It showed how to integrate different ray casting kernels into already existing OpenGL frameworks. The interface and the example program did suffice for the SMS' visualization group to integrate the ray casting kernels into GeoDict without further major help (Figure 2.10).

After successful integration the SMS used the render kernels as they where intended to be used, i.e., display flow fields and large data sets (oil filtration simulation in 2.7.2, Figure 2.11).

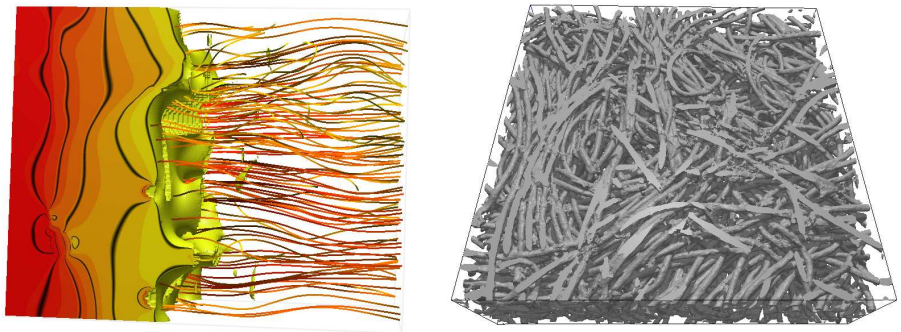

Figure 2.11: Left: A flow field visualized by the advanced iso surface ray caster and some streamlines displayed using OpenGL. Right: A large CT-scan of fibres.

But soon the SMS used the fact that the ray casting kernels are not only OpenGL compatible, but also compatible to each other. They showed different applications where they merged both ray casting kernels into one image (Figure 2.12). This was a positive surprise. The hybrid viewing technique was intended to support that different ray casting kernels may be merged together with OpenGL into one image, but that feature has actually neither been tested before, nor was this possibility mentioned to the SMS. The fact that this did work without any modifications and without the need of mentioning shows, that the hybrid viewer is well and intuitively designed. The positive feedback of the SMS does emphasize this even further.

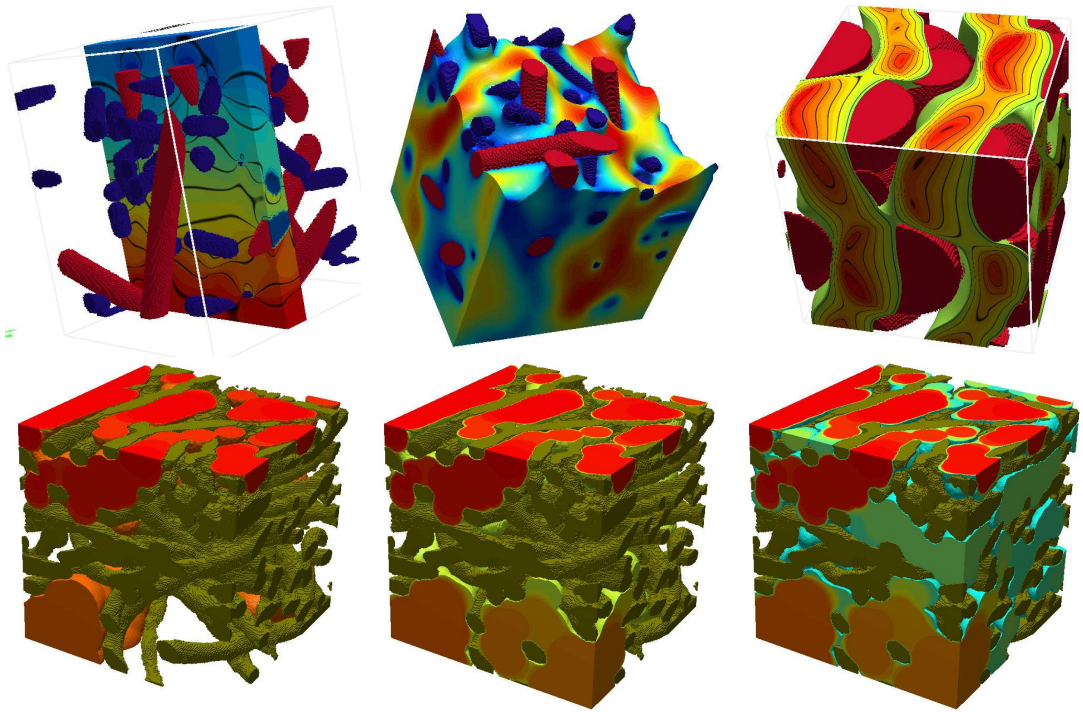

Figure 2.12: Different cases where two ray casters (microstructure and flow field) are merged into one image (top left/right additionally together with OpenGL).

2.9 Future Research

Even though the focus in this chapter is on the interactive visualization of scientific data sets, the hybrid visualization technique is not restricted to that. The basic concept also fits for gaming applications. Many games visualize player models that are dynamically moving through a mostly static environment. Highly complex static environments may be ray cast instead of rasterized. This frees resources from the graphics card, which may be used to visualize more complex player models.

In the setup described here three rasterization passes are used, where two passes are needed for rendering the scene once in shadow and once not. High end graphics card do support multiple render targets. The two images could be written in one pass, discarding one rasterization pass.

Chapter 3

Implicit *k*d-trees

3.1 Abstract

New implicit *k*d-trees are introduced here. They have fast construction times, low memory requirements, and allow on today's desktop computers interactive iso surface ray casting and maximum intensity projection (MIP) of data sets that are larger than one half of the machine's given main memory. Iso surface, clip planes, colors and the visualization technique (iso surfacing with or without shadows, MIP) can be changed interactively, since they all are computed on the fly. Due to the logarithmic dependency between visualization time and scene size it is possible to visualize even massively complex data sets at interactive framerates.

3.2 Problem

Many scientific data sets are scalar fields $S \in \mathbb{R}^3$ ($\in \mathbb{R}^4$) (e.g., CT- and MRI-scans, seismic- and CFD data sets) which are often given as 3D (or 4D: 3D + time) rectilinear grids with dimensions dim_x, dim_y, dim_z(, dim_t) discrete values $s_{i,j,k(,u)}$ and grid planes $p_x(i)$, $p_y(j)$, $p_z(k)$ (and times $t(u)$). These data sets are often massively complex and are steadily increasing in size.

Iso surface ray tracing and MIP accelerated through min/max *k*d-trees have a logarithmic dependency between visualization time and scene size, making them

ideal techniques for the visualization of massively complex scalar fields. But explicitly stored *k*d-trees require a multiple of the original data set's memory which strongly decreases the maximal possible size of a data set that can be rendered at interactive framerates.

3.3 State of the Art

This section's focus is on software based visualization methods, which achieve interactive framerates for at least medium sized data sets (512^3 voxels) on standard desktop machines.

Interactive iso surface raytracing (and MIP) for large regular scalar fields through software has first been achieved by Parker et al. (38) ((39)) by a brute-force implementation on a high end (distributed) shared-memory multiprocessor system. DeMarle et al. ported this system to run on PC clusters (9), (8). A major step forward to interactive iso surface ray tracing on *desktop* machines has recently been achieved by Wald et al. (50), where for each scalar field a min/max *k*d-tree is constructed, which is used to accelerate ray traversal of regions not containing an iso surface.

The results of Wald et al.'s work show that state of the art shared memory desktop machines are able to visualize large data sets at fairly interactive frame rates. But their min/max *k*d-trees have the drawbacks of being at least twice (and up to 16 times) as large as the scalar field itself, which is a too strong memory overhead for the visualization of large data sets.

Interactive high-quality MIP through software on desktop machines has first been achieved by Mroz et al. (32). Mora et al. recently implemented a fast object order ray casting algorithm (31), (30). Mora et al. also proved mathematically that MIP through image order ray tracing has for discretized gray levels an average visualization time of order $O(lg(n))$ (30). For accelerating empty space skipping Mora et al. used octrees similar to Wald et al.'s min/max *k*d-trees, with the major difference that Mora et al.'s trees did also store leaf nodes, which therefore have a memory overhead of more than twice the scalar field's size. They also tried to implement an image order ray casting algorithm, but stated: "This result was not sufficient to get interactive rendering rates" without

supplying any framerates. Both MIP visualization techniques are fairly interactive for medium sized data sets. But both visualization techniques have memory overheads of more than twice the data's size, and a visualization time of at least order $O(n^{2/3})$. They are therefore unsuited for visualizing large data sets. Both visualization techniques are furthermore tied to parallel projection and do not support perspective projection.

3.4 Results

The different contributions of this chapter are:

- A new and very general definition for implicit *k*d-trees together with their construction and traversal algorithms is introduced.
- Corresponding implicit bit, bitmask, and min/max *k*d-trees are defined.
- An efficient memory reduction scheme for (implicit) min/max *k*d-trees is introduced. The resulting optimized implicit min/max or max *k*d-trees require as much resp. half as much memory as the original scalar field.
- Besides the fact that optimized implicit max *k*d-trees may be used for accelerated semi iso surfacing, it is shown that those trees may also be used for accelerating maximum intensity projection.
- It is shown that the visualization time using implicit *k*d-trees is comparable to the visualization times using explicit *k*d-trees.
- It is shown that the construction of implicit *k*d-trees is linear, scales linear to the amount of processor cores used, and is fairly fast.

3.5 Definition

The definition for implicit *k*d-trees is given here. It is emphasized that this definition and the corresponding implicit *k*d-trees evolving out of this definition are new. They have very simple and general construction- and traversal-algorithms which are then modified and used for different visualization tasks.

An **implicit *k*d-tree** is a *k*d-tree defined implicitly above a rectilinear grid. Its splitting planes' positions and orientations are not given explicitly but implicitly by some recursive splitting-function defined on the hyperrectangles belonging to the tree's nodes. Each inner node's split plane is positioned on a grid plane of the underlying grid, partitioning the node's grid into two subgrids.

3.6 Nomenclature

The terms "min/max *k*d-tree" and "implicit *k*d-tree" are sometimes mixed up. This is due to the fact that the first publication using the term "implicit *k*d-tree" (50) did actually use explicit min/max *k*d-trees but referred to them as "implicit *k*d-trees" to indicate that they may be used to ray trace implicitly given iso surfaces. Nevertheless that publication used also slim *k*d-trees which are a subset of the implicit *k*d-trees with the restriction that they can only be built over integer hyperrectangles with sidelengths that are powers of two.

As it is possible to assign attributes to implicit *k*d-tree nodes, one may refer to an implicit *k*d-tree which has min/max values assigned to its nodes as an "implicit min/max *k*d-tree".

3.7 Construction

Implicit *k*d-trees are in general not constructed explicitly. When accessing a node, its splitting plane orientation and position are evaluated using a specific splitting function defining the tree.

3.7.1 Splitting Functions

Splitting functions define whether a node in an implicit *k*d-tree is an inner node, and if so, what orientation and position the corresponding split plane has. Splitting functions may be adapted to special purposes. Different splitting functions may result in different trees for the same underlying grid. Below are two specifications of special splitting function classes.

- A **non-degenerated splitting function** does not allow the creation of degenerated nodes (nodes whose corresponding integer hyperrectangle's volume is equal zero). The corresponding implicit kd-tree is a **non-degenerated implicit *k*d-tree** which is also a full binary tree, i.e., it has for n leaf nodes $n-1$ inner nodes.

- A **complete splitting function** is a non-degenerated splitting function that always splits tree nodes that are not single grid cells. Its corresponding implicit kd-trees are **complete implicit *k*d-trees** whose leaf nodes are single grid cells such that those trees have always one inner node less than the amount of gridcells given in the grid.

A complete splitting function is for example the **grid median splitting function**. It creates fairly balanced implicit kd-trees by using k-dimensional integer hyperrectangles $hyprec[2][k]$ which belong to each node of the implicit kd-tree. The hyperrectangles define which grid cells of the rectilinear grid belong to their corresponding node. If the volume of this hyperrectangle equals one, the corresponding node is a single grid cell and is therefore not further subdivided and marked as leaf node. Otherwise the hyperrectangle's longest extend is chosen as orientation o, where if the hyperrectangle has multiple maximal side lengths the first corresponding dimension is chosen. The corresponding split plane p is positioned onto the grid plane that is closest to the hyperrectangle's grid median along that orientation.

Algorithm 2 Grid Median Splitting Function

if (volume($hyprec$) > 1) **then**
  $o = min\{argmax_{i=1\ldots k}(hyprec[1][i] - hyprec[0][i])\}$
  $p = \lfloor (hyprec[0][o] + hyprec[1][o])/2 \rfloor$

Where volume($hyprec$) $\widehat{=}$ $\prod_{i=0}^{k-1}(hyprec[1][i] - hyprec[0][i])$

In the remainder of this chapter the grid median splitting function and a slight modification of it (the even grid median splitting function) is used.

3.7.2 Assigning Attributes to Inner Nodes

An obvious advantage of implicit kd-trees is that their splitting plane's orientations and positions need not to be stored explicitly.

But some applications require besides the splitting planes' orientations and positions further attributes at the inner tree nodes. These attributes may be for example single bits or single scalar values, defining whether the sub grids belonging to the nodes are of interest or not. For complete implicit kd-trees it is possible to pre-allocate a correctly sized array of attributes and to assign each inner node of the tree to a unique element in that allocated array.

The amount of gridcells in the grid is equal the volume of the integer hyperrectangle belonging to the grid. As a complete implicit kd-tree has one inner node less than grid cells $n_{inner} = \text{volume}(hyprec[2][k]) - 1$, it is known in advance how many attributes need to be stored. This relation together with the complete splitting function defines a recursive formula which allows to assign to each splitting plane a unique element in the allocated array. The corresponding formula is given in Algorithm 3.

Algorithm 3 Assigning Attributes to Inner Nodes of a Complete Implicit kd-tree

variables global k; $dim[k]$; $attributes[(\prod_{i=0}^{k-1} dim[i]) - 1]$;
variables input $pos = 0$; $hyprec[2][k] = \{\{0, \ldots, 0\}, \{dim[0], \ldots, dim[k-1]\}\}$;
variables local $hyprec_l[2][k]$; $hyprec_r[2][k]$; $attributes_l$; $attributes_r$; o; p;
  function assign(pos, $hyprec$)
    if $(volume(hyprec) \equiv 1)$ **then**
      return get($hyprec$);
    else
      $o = min\{argmax_{i \in \{0,\ldots k-1\}}(hyprec[1][i] - hyprec[0][i])\}$;
      $p = \lfloor (hyprec[0][o] + hyprec[1][o])/2 \rfloor$;
      $hyprec_l = hyprec$; $hyprec_l[1][o] = p$; $hyprec_r = hyprec$; $hyprec_r[0][o] = p$
      $attribute_l = \text{assign}(pos + 1, hyprec_l)$;
      $attribute_r = \text{assign}(pos + volume(hyprec_l), hyprec_r)$;
      return merge($attributes_l, attributes_r, attributes[pos]$);

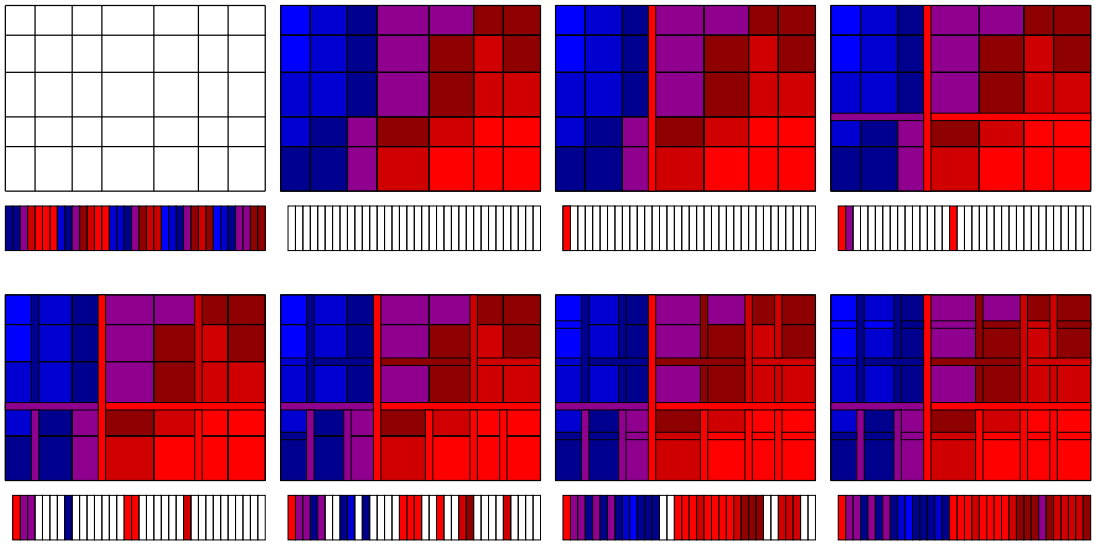

Figure 3.1: Construction of an implicit max kd-tree above a rectilinear grid. The first image shows the grid where underneath the corresponding array holding its scalar values is shown. The second image shows the grid's voxels colored in the intensities of their respective scalar values, where underneath the pre-allocated array for the corresponding implicit max kd-tree is given (note that it requires one memory position less than the array for the scalar field). Each following image shows the split planes (colored in the same intensity than its corresponding max value) of one additional level of the underlying implicit max kd-tree, where underneath the storage of the corresponding max values in the pre-allocated array is given.

It is worth mentioning that this algorithm works for all rectilinear grids. The corresponding integer hyperrectangle does not necessarily have to have side-lengths that are powers of two (Figure 3.1). Furthermore, it is not required to store the memory locations of the constructed trees explicitly, such that they may be used on both 32 bit and 64 bit machines. Note that the construction time is of order $\mathcal{O}(kn)$ and therefore linear to the volume n for a fixed dimension k.

3.8 Traversal

Traversal of implicit kd-trees is similar to the traversal of explicit kd-trees (46). The differences are: The check of the current node's attribute that specify if the underlying rectilinear grid contains regions of interest (and if not, the current node is not further traversed but skipped), the "on the fly" computations (the split plane's orientation and position, the children's hyperrectangles and memory locations), the split plane look up, and the additional stacking operations (the backside node's hyperrectangle and attribute). A generalized version of the traversal algorithm for k dimensions is given in Algorithm 4.

The initialization of the traversal algorithm is also given, where the integer hyperrectangle $hyprec[2][k]$ and the help variables $R_{1/d}[k]$ and $flags[k]$ are initialized. The hyperrectangle is during each traversal step updated and is used to define the position of an inner node's attribute or to define a leaf node's position inside the rectilinear grid. The help variables are first used to perform a fast and efficient clipping against the axis aligned clip box $clipBox[2][k]$ (which is contained in or equal sized as the axis aligned bounding box $boundingBox[2][k]$, where $boundigBox[0][i] = planes[i][0]$; $boundingBox[1][i] = planes[i][dim[i]]$) and are then in the traversal re-used to replace costly divisions by simple multiplications and to replace costly conditional operations by simple data look ups respectively.

The traversal algorithm given here is kept general enough to serve rectilinear grids. It may slightly be optimized for regular or cartesian grids where the split plane look up may be replaced by a simple scaling operation $d = (scale[o]p - R_o[o])R_{1/d}[o]$ or entirely discarded $d = (p - R_o[o])R_{1/d}[o]$ respectively. Tree construction is independent of any of those three grid types and stays unmodified. Note that it is possible to change the grid planes' positions/scale (i.e. deform the rectilinear grid to some other rectilinear grid), where neither the tree needs to be reconstructed, nor the traversal algorithm needs to be changed. It suffices to simply update $planes[k][dim[k]-1]$ or $scale[k]$.

In the remainder of this chapter the focus is on $k = 3$ where the hyperrectangle is a simple axis aligned integer box $box[2][3]$ and with the correlations $dim[0] \mathrel{\widehat{=}} dim_x$, $dim[1] \mathrel{\widehat{=}} dim_y$, $dim[2] \mathrel{\widehat{=}} dim_z$, and $planes[0] \mathrel{\widehat{=}} p_x$, $planes[1] \mathrel{\widehat{=}} p_y$, $planes[2] \mathrel{\widehat{=}} p_z$. The corresponding implementations have been restricted to and optimized for $k = 3$.

Algorithm 4 Traversal of an Implicit kd-tree

variables global k; $dim[k]$; $planes[k][dim[k]+1]$; $attributes[\prod_{i=0}^{k-1} dim[i]-1]$; $clipBox[2][k]$;
variables input $R_o[k]$; $R_d[k]$; d_n; d_f;
variables local $n = d_n$; $f = d_f$; $R_{1/d}[k]$; $hyprec[2][k]$; $flags[k]$; $flag$;
$pos = 0$; $pos_{l/r}[2]$; $offset$; p; o; tmp; i;

```
for (i = 0, ..., k − 1)
    hyprec[0][i] = 0; hyprec[1][i] = dim[i]; R_{1/d}[i] = 1/R_d[i];
    if(R_d[i] > 0) then flags[i] = 1; else flags[i] = 0;
    n = max((clipBox[1 − flags[i]][i] − R_o[i])R_{1/d}[i], n);
    f = min((clipBox[flags[i]][i] − R_o[i])R_{1/d}[i], f);
if(n > f) then return NoHit;
loop
    o = min(argmax_{i∈{0,...k−1}}(hyprec[1][i] − hyprec[0][i])); flag = flags[o];
    while ((hyprec[1][o] − hyprec[0][o]) > 1∧ isImportant(attributes[pos]))
        p = ⌊(hyprec[0][o] + hyprec[1][o])/2⌋; d = (planes[o][p] − R_o[o])R_{1/d}[o];
        offset = (p − hyprec[0][o]) ∏_{i∈{0,...,k−1}:i≠o} (hyprec[1][i] − hyprec[0][i]);
        pos_{l/r}[0] = pos + 1; pos_{l/r}[1] = pos + offset;
        if (d < n) then
            hyprec[1 − flag][o] = p; pos = pos_{l/r}[flag];
        else if (d > f) then
            hyprec[flag][o] = p; pos = pos_{l/r}[1 − flag];
        else
            tmp = hyprec[1 − flag][o]; hyprec[1 − flag][o] = p
            pushOnStack(hyprec, d, f, pos_{l/r}[flag]);
            hyprec[1 − flag][o] = tmp; hyprec[flag][o] = p;
            pos = pos_{l/r}[1 − flag]; f = d;
        o = min(argmax_{i∈{0,...k−1}}(hyprec[1][i] − hyprec[0][i])); flag = flags[o];
    if ((hyprec[1][o] − hyprec[0][o]) ≡ 1) then
        if (intersect(R_o, R_d, n, f, hyprec)) then
            d_f = f; return Hit;
    if(stackIsEmpty()) then return NoHit;
    popFromStack(hyprec, n, f, pos);
```

3.9 Implicit Bit *k*d-trees

The most simple attribute that may be assigned to an implicit *k*d-tree is a single bit specifying if the underlying sub grid which belong to the inner node contains regions of interest. Those implicit bit *k*d-trees are especially suited to visualize **boolean scalar fields**, i.e. scalar fields that contain only zeros and ones (Figure 3.2). Boolean scalar fields may be stored very efficiently in a bitfield such that they require only one bit per voxel. An implicit bit *k*d-tree built over such a field requires only the same amount of memory as the bitfield does.

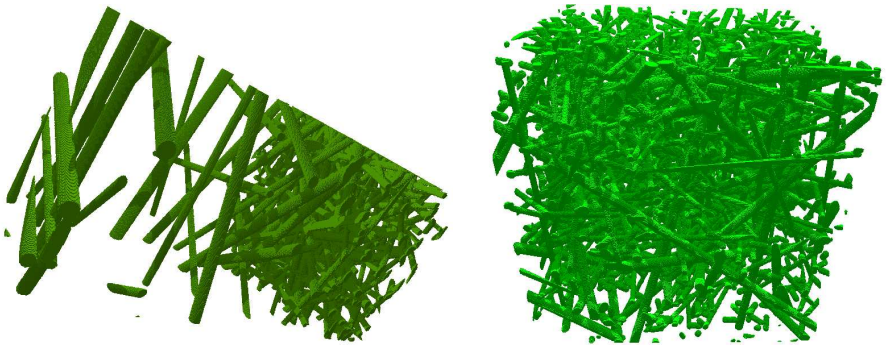

Figure 3.2: Boolean scalar fields visualized using implicit bit *k*d-trees.

Tree construction and traversal are identical as the procedures given in Algorithms 3 and 4, where the definitions for the three functions *get*, *merge* and *isImportant* are given in Algorithm 5.

Implicit bit *k*d-trees are not restricted to the visualization of boolean scalar fields. They may also be built over general scalar fields, where the bit may for example specify if the underlying sub grids contain scalar values of interest or an iso surface to a pre-defined scalar value.

Algorithm 5 Function Definitions for Implicit Bit *k*d-trees

```
function get(box[2][3])
  return bitField[(box[0][2]dim[1] + box[0][1])dim[0] + box[0][0]];
function merge(bit_l, bit_r, bit)
  bit = bit_l ∨ bit_r;
  return bit;
function isImportant(bit)
  return bit ≠ 0;
```

3.10 Implicit Bitmask *k*d-trees

The generalization to implicit bit *k*d-trees are implicit bitmask *k*d-trees, where the attributes assigned to the nodes are not single bits, but bitmasks.

The advantage of implicit bitmask *k*d-trees is that the user may interactively choose some importance bitmask. The corresponding voxels are directly displayed where the implicit bitmask *k*d-tree needs not to be rebuild (Figure 3.3).

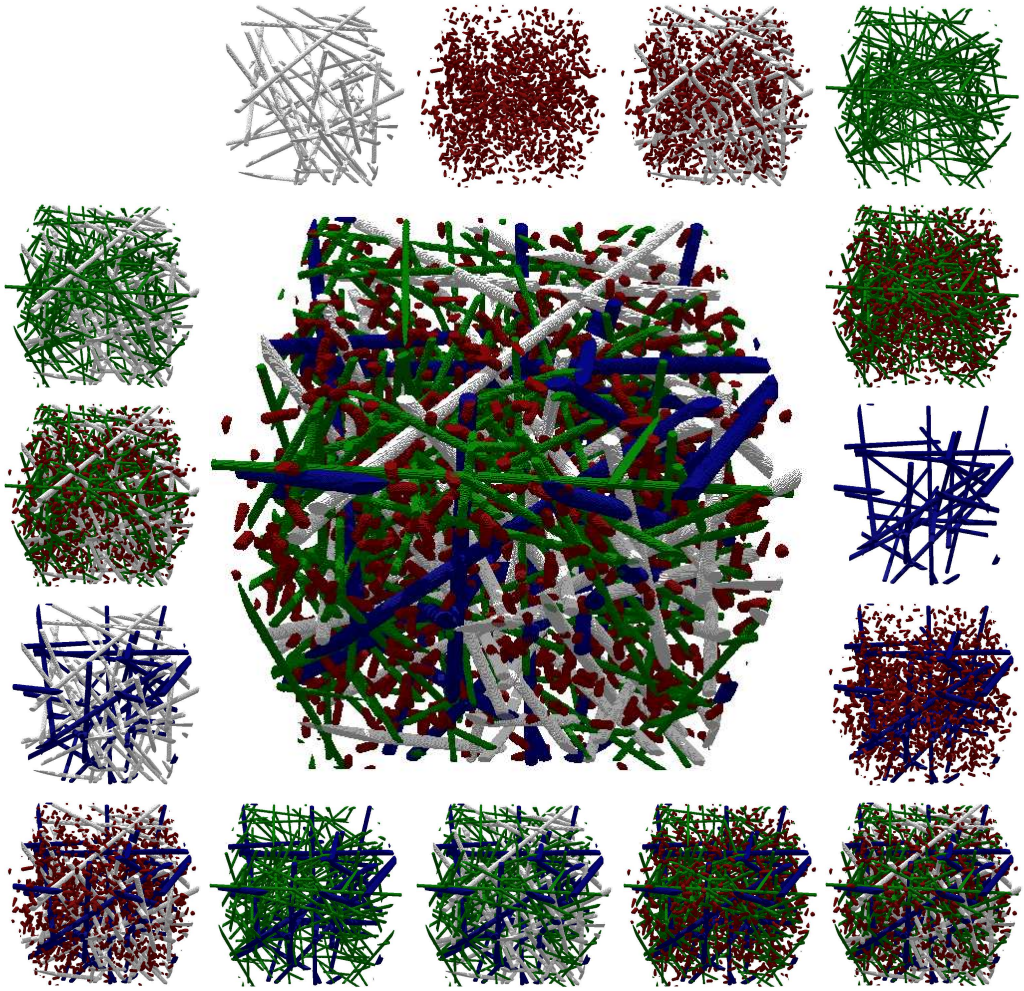

Figure 3.3: A 4-bit scalar field (center) visualized using a 4-bit implicit bitmask *k*d-tree. The data set is a filter consisting of four different kind of fibres, each represented by a single bit. The visualizations to the 16 possible bitmasks (from top left: 0 to bottom right: 15) are shown around the main image.

The three functions *get*, *merge* and *isImportant* are similar as in Algorithm 5, with the difference, that the operations are performed not on single bits, but on bitmasks. The function *get* returns a bitmask, the function *merge* applies the OR operation to bitmasks, and the function *isImportant* applies the AND operation of the bitmask to the given importance bitmask and checks the result for unequality to zero.

3.11 Implicit Min/Max *k*d-trees

Other attributes that may be assigned to implicit *k*d-trees are s_{min}/s_{max} values specifying the minimal/maximal scalar values given in the underlying sub grids belonging to the inner nodes. This allows to apply accelerated (semi) iso surfacing for different iso surfaces, which is an improvement compared to implicit bit *k*d-trees (which may only be used to visualize one iso surface given by a pre-defined scalar value). This comes for the price of higher storage requirements as now not a single bit but two scalar values have to be stored per inner node.

Iso surface visualization works on grid cells (and not only on single scalar values), which are for rectilinear grids axis-aligned cuboids with a scalar value assigned to each of its eight corners (instead of single voxels equipped with one scalar value). A scalar field with $dim_x \times dim_y \times dim_z$ scalar values has $n_{leaf} = (dim_x-1)(dim_y-1)(dim_z-1)$ grid cells which correspond to the implicit min/max *k*d-tree's leaf nodes.

Storing for each inner node its corresponding s_{min} and s_{max} values will result in $2n_{leaf} - 2$ scalar values (roughly twice as large as the corresponding scalar field). As leaf nodes are not single scalar values but grid cells equipped with eight scalar values, it may be considered to assign min/max values to leaf nodes. But this results in storing $4n_{leaf} - 2$ scalar values (roughly four times as large as the corresponding scalar field) and is not done in order to minimize the memory overhead. This comes for the price that for a leaf node that is entered by some rays, the eight corresponding scalar values have to be read from the scalar field. These values define, if an iso surface may exist in that leaf or not. Even though this is a bit more expensive then the simple reading of a min/max value, the

visualization speed is not affected significantly, as the traversal of leaf nodes is much less common than the traversal of inner nodes.

Tree construction and traversal of implicit min/max *k*d-trees are almost identical to the procedures given in Algorithms 3 and 4. Only the initialization differs slightly, as the grid planes lie not between but on the scalar values such that exactly dim_x, dim_y, dim_z grid planes are given (and not $dim_x{+}1$, $dim_y{+}1$, $dim_z{+}1$) which results in a smaller box $box[2][3] = \{\{0,0,0\}, \{dim_x{-}1, dim_y{-}1, dim_z{-}1\}\}$ and different amount of leaf nodes n_{leaf}, inner nodes $n_{inner} = n_{leaf} - 1$ and attributes $attributes[n_{inner}]$. The definitions for the three functions *get*, *merge* and *isImportant* are given in Algorithm 6.

Algorithm 6 Function Definitions for Implicit Min/Max *k*d-trees

function get($box[2][3]$)
  $\Omega = \cup_{i,j,k\in\{0,1\}} s_{box[i][0],box[j][1],box[k][2]}$
  return $\min(\Omega)/\max(\Omega)$;
function merge($s_{min,l}/s_{max,l}$, $s_{min,r}/s_{max,r}$, s_{min}/s_{max})
  $s_{min}/s_{max} = \min(s_{min,l}, s_{min,r})/\max(s_{max,l}, s_{max,r})$;
  return s_{min}/s_{max};
function isImportant(s_{min}/s_{max})
  return $s_{min} \leq s_{iso} \leq s_{max}$;

The *isImportant* function given here is the one used for iso surfacing. For semi iso surfacing the function returns $s_{min} \leq s_{iso}$ /$s_{iso} \leq s_{max}$.

3.12 Optimized Implicit Min/Max *k*d-trees

The implicit min/max *k*d-trees given in Section 3.11 have the disadvantage that they require to store roughly twice the amount of scalar values than given in the corresponding scalar field. The resulting additional memory overhead of roughly twice the data size is quite high, especially when aiming at the visualization of large data sets. The goal is to further compress implicit min/max *k*d-trees where construction- and traversal-performance shall not be decelerated significantly.

For slim *k*d-trees it has been suggested to discretize the s_{min} and s_{max} down to e.g., 4 bits (50) where that possible compression scheme was not investigated. The information loss of for example either 4 or even 12 bits (for 8 bit rsp. 16 bit resolution data sets) would surely affect visualization performance negatively, as rays will traverse regions not containing iso surfaces due to the badly discretized s_{min} and s_{max} values.

A different and new compression scheme is introduced here. It is by far more precise as the compression scheme suggested above, as only one single bit of information is lost, independent of the underlying scalar field's bit resolution: The observation that s_{max} of a parent node coincides with either its left or its right child nodes' maximal value $s_{max,l}$, $s_{max,r}$ is used to further compress the implicit *k*d-trees.

If s_{max} is already given for a parent node, one further maximal value of a child $s_{max,c}$ and a flag defining to which child $s_{max,c}$ belongs are enough for evaluating both children's s_{max} values. This flag may be stored in the first bit of the $s_{max,c}$ value, which therefore needs to be discretized by rounding it up to an odd number. The same principle also works for $s_{min,c}$ by rounding it down to an even number.

In the traversal technique described above in 3.11, the s_{min} and s_{max} values of an inner node have to be read when a ray enters a node. By using the new observation it is possible to know very fine approximations of s_{min}/s_{max} in advance, before traversing a node. This first minimizes memory look ups and second allows to store one level of nodes less, which correlates to a memory saving of one half, i.e, the tree requires to store roughly the same amount of scalar values than given in the corresponding scalar field.

The compression scheme only works for binary trees which the min/max *k*d-trees described above in 3.11 are not generally (since only the inner nodes are stored). The data resolution is therefore virtually padded to odd sizes (resulting in boxes $box[2][3]$ with even side lengths) and the grid median splitting function is slightly modified. During tree construction and traversal the split plane will be round down to an even number, as long as the longest side of the box is greater than two. This forces the tree of inner nodes to be binary.

Algorithm 7 Even Grid Median Splitting Function

Require: $\bigwedge_{i=i}^{2}(box[1][i] - box[0][i] \equiv 1 \vee ((box[1][i] - box[0][i]) \bmod 2) \equiv 0)$;
  if (volume(box) > 1) **then**
    $o = \min(argmax_{i\in\{0,1,2\}}(box[1][i] - box[0][i]))$;
    $p = box[0][o] + box[1][o]$; **if**$(p > 2)$ **then** $p = \lfloor p/2 \rfloor_2$ **else** $p = p/2$

Where $\lfloor \bullet \rfloor_2$ rounds down to the next even number: $\lfloor x \rfloor_2 \mathrel{\widehat{=}} 2\lfloor x/2 \rfloor$.

The resulting tree construction and tree traversal (which are modifications of the algorithms described in 3.12) are given in Algorithms 8 and 9. The definitions for the three corresponding functions *get*, *merge* and *isImportant* are given in Algorithm 10.

Algorithm 8 Assigning Attributes to an Optimized Implicit Min/Max *k*d-tree

variables global $dim[3]$; $s_{min/max}[\prod_{i=0,}^{2} \lfloor dim[i] \rfloor_2]$;
variables input $pos = 2$; $box[2][3] = \{\{0,0,0\}, \{\lfloor dim[0] \rfloor_2, \lfloor dim[1] \rfloor_2, \lfloor dim[2] \rfloor_2\}\}$;
variables local $box_l[2][3]$; $box_r[2][3]$; s_{min}; s_{max}; $s_{min,l}$; $s_{min,r}$; $s_{max,l}$; $s_{max,r}$; o; p;
  $s_{min/max}[0] = \min(S)$; $s_{min/max}[1] = \max(S)$;
  function assign(pos, box)
    if (volume(box)$\equiv 2$) **then**
      $o = argmax_{i\in\{0,1,2\}}(box[1][i] - box[0][i])$; $p = (box[0][o] + box[1][o])/2$;
      $box_l = box$; $box_l[1][o] = p$; $box_r = box$; $box_r[0][o] = p$
      $s_{min,l}/s_{max,l} = \text{get}(box_l)$; $s_{min,r}/s_{max,r} = \text{get}(box_r)$;
      return merge($s_{min,l}/s_{max,l}$, $s_{min,r}/s_{max,r}$, s_{min}/s_{max});
    else
      $o = \min(argmax_{i\in\{0,1,2\}}(box[1][i] - box[0][i]))$;
      $p = box[0][o] + box[1][o]$; **if**$(p > 2)$ **then** $p = \lfloor p/2 \rfloor_2$ **else** $p = p/2$
      $box_l = box$; $box_l[1][o] = p$; $box_r = box$; $box_r[0][o] = p$
      $s_{min,l}/s_{max,l} = \text{assign}(pos + 2, box_l)$;
      $s_{min,r}/s_{max,r} = \text{assign}(pos + volume(box_l), box_r)$;
      return merge($s_{min,l}/s_{max,l}$, $s_{min,r}/s_{max,r}$, $s_{min/max}[pos]/s_{min/max}[pos + 1]$)

Algorithm 9 Traversal of an Optimized Implicit Min/Max *k*d-tree

variables $s_{min/max}[\prod_{i=0}^{2} \lfloor dim[i] \rfloor_2]$; s_{min}; s_{max}; $s_{min,l/r}[2]$; $s_{max,l/r}[2]$; ...
$box[2][3] = \{\{0,0,0\}, \{\lfloor dim[0] \rfloor_2, \lfloor dim[1] \rfloor_2, \lfloor dim[2] \rfloor_2\}\}$;
$s_{min} = s_{min/max}[0]$; $s_{max} = s_{min/max}[1]$; $pos = 2$;
loop
$o = \min(argmax_{i \in \{0,1,2\}}(box[1][i] - box[0][i]))$; $flag = flags[o]$;
while $((box[1][o] - box[0][o]) > 1 \wedge$ isImportant$(s_{min}/s_{max}))$
$p = box[0][o] + box[1][o]$; **if**$(p > 2)$ **then** $p = \lfloor p/2 \rfloor_2$ **else** $p = p/2$
$d = (planes[o][p] - R_o[o])R_{1/d}[o]$;
$offset = (p - box[0][o]) \prod_{i \in \{0,1,2\}: i \neq o} (box[1][i] - box[0][i])$;
$pos_{l/r}[0] = pos + 2$; $pos_{l/r}[1] = pos + offset$;
$s_{min,l/r}[0] = s_{min,l/r}[1] = s_{min}$; $s_{max,l/r}[0] = s_{max,l/r}[1] = s_{max}$;
if $(offset > 1)$ **then**
$s_{min,l/r}[1 - (s_{min/max}[pos] \bmod 2)] = \lfloor s_{min/max}[pos] \rfloor_2$;
$s_{max,l/r}[1 - (s_{min/max}[pos+1] \bmod 2)] = \lfloor s_{min/max}[pos+1] \rfloor_2 + 1$;
if $(d < n)$ **then**
$box[1 - flag][o] = p$; $pos = pos_{l/r}[flag]$;
$s_{min} = s_{min,l/r}[flag]$; $s_{max} = s_{max,l/r}[flag]$;
else if $(d > f)$ **then**
$box[flag][o] = p$; $pos = pos_{l/r}[1 - flag]$;
$s_{min} = s_{min,l/r}[1 - flag]$; $s_{max} = s_{max,l/r}[1 - flag]$;
else
$tmp = box[1 - flag][o]$; $box[1 - flag][o] = p$
pushOnStack$(box, d, f, pos_{l/r}[flag], s_{min,l/r}[flag], s_{max,l/r}[flag])$;
$box[1 - flag][o] = tmp$; $box[flag][o] = p$; $pos = pos_{l/r}[1 - flag]$;
$s_{min} = s_{min,l/r}[1 - flag]$; $s_{max} = s_{max,l/r}[1 - flag]$; $f = d$;
$o = \min(argmax_{i \in \{0,1,2\}}(box[1][i] - box[0][i]))$; $flag = flags[o]$;
if $((box[1][o] - box[0][o]) \equiv 1 \wedge \bigwedge_{i=0}^{2} box[1][i] < dim[i])$ **then**
if (intersect(R_o, R_d, n, f, box)) **then**
$d_f = f$; **return** Hit;
if(stackIsEmpty()) **then return** NoHit;
popFromStack$(box, n, f, pos, s_{min}, s_{max})$;

Algorithm 10 Function Definitions for Optimized Implicit Min/Max *k*d-trees

function get($box[2][3]$)
  $s_{min}/s_{max} = \max(S)/\min(S)$;
  if $(\bigwedge_{i=0}^{2} box[1][i] < dim[i])$ **then**
    $\Omega = \cup_{i,j,k\in\{0,1\}} s_{box[i][0],box[j][1],box[k][2]}$
    $s_{min}/s_{max} = \min(\Omega)/\max(\Omega)$;
  return s_{min}/s_{max};
function merge($s_{min,l}/s_{max,l}$, $s_{min,r}/s_{max,r}$, s_{min}/s_{max})
  if $(s_{min,l} < s_{min,r})$ **then** $s_{min} = \lfloor s_{min,r} \rfloor_2$ **else** $s_{min} = \lfloor s_{min,l} \rfloor_2 + 1$;
  if $(s_{max,l} > s_{max,r})$ **then** $s_{max} = \lfloor s_{max,r} \rfloor_2$ **else** $s_{max} = \lfloor s_{max,l} \rfloor_2 + 1$;
  return $\lfloor \min(s_{min,l}, s_{min,r}) \rfloor_2 / \lfloor \max(s_{max,l}, s_{max,r}) \rfloor_2 + 1$;
function isImportant(s_{min}/s_{max})
  return $s_{min} \leq s_{iso} \leq s_{max}$;

3.13 Optimized Implicit Max *k*d-trees

It is also possible to assign only s_{max} values to implicit *k*d-trees in the same manner as described above in 3.12 resulting in optimized implicit max *k*d-trees. Discarding the storage of the s_{min} values results in further memory savings, where the tree requires to store at most half the amount of scalar values than given in the corresponding scalar field. This comes for the price of loosing the ability to perform accelerated iso surfacing. Accelerated semi iso surfacing is still possible for $s_{iso} \leq s_{max}$.

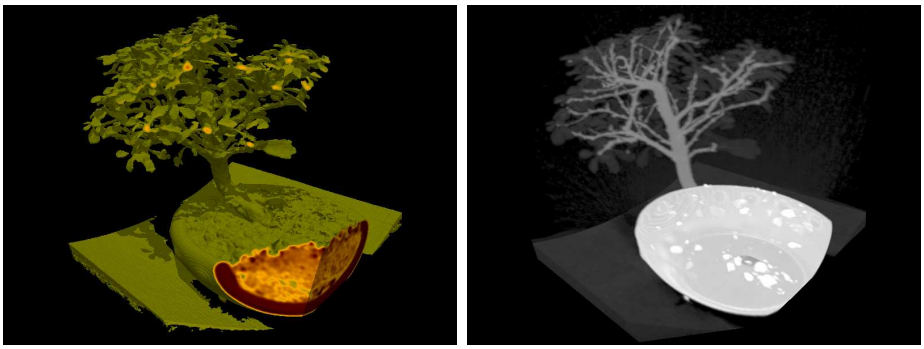

Figure 3.4: Scalar field visualized by semi iso surfacing and MIP using optimized implicit max *k*d-trees.

It is furthermore possible to perform accelerated maximum intensity projection (MIP), where the corresponding traversal algorithm differs only slightly from the traversal code for iso surface visualization. In the traversal code the current maximum density s_{MIP} is initialized with the ray's enter point of the scene. Inner nodes will only be traversed if their s_{max} value is greater than s_{MIP}. Leaf intersection does not terminate the traversal, but update s_{MIP}.

Algorithm 11 MIP Traversal of an Optimized Implicit Max *k*d-tree

variables $s_{maxima}[(\prod_{i=0}^{2} \lfloor dim[i] \rfloor_2)/2]$; $d_{n/f}[2] = \{d_n, d_f\}$; $flag_{MIP}$; s_{MIP}; ...
⋮
  $s_{MIP} = \text{getValue}(R_o + d_{n/f}[0]R_d)$;
  loop
    ...
    while $((box[1][o] - box[0][o]) > 1 \wedge (s_{MIP} < s_{max}))$
      ...
      $pos_{l/r}[0] = pos + 1$; $pos_{l/r}[1] = pos + offset/2$;
      ...
      if $(d < d_{n/f}[0])$ **then** ... **else if** $(d > d_{n/f}[1])$ **then** ...
      else
        $flag_{MIP} = 1$;
        if $(s_{max,l/r}[0] < s_{max,l/r}[1])$ **then** $flag_{MIP} = 0$ **else** $flag = 1 - flag$;
        ... $tmp = d_{n/f}[flag_{MIP}]$; $d_{n/f}[flag_{MIP}] = d$;
        pushIntoQueue$(box, d_{n/f}[0], d_{n/f}[1], pos_{l/r}[flag], s_{max,l/r}[flag])$;
        $d_{n/f}[flag_{MIP}] = tmp$; $d_{n/f}[1 - flag_{MIP}] = d$;
⋮
    if $((box[1][o] - box[0][o]) \equiv 1 \wedge \bigwedge_{i=0}^{2} box[1][i] < dim[i])$ **then**
      intersect$(R_o, R_d, n, f, box, s_{MIP})$
⋮

For MIP, the final image is not affected by the order in which the tree's nodes are traversed. The nodes are therefore not traversed in "nearest first" but in "highest maximum intensity first" order: When a ray traverses both children of an inner node, not the backside child, but the child with lower maximum

intensity is pushed onto the stack. The traversal stack will therefore be used as a "priority queue" (see (39)). Nodes that are popped from the traversal stack are automatically tested, if their maximum density exceeds the current maximum density. If that's not the case, they will be discarded. Tree traversal therefore converges to the global maximum density along the ray's path. The corresponding traversal algorithm (a modification of Algorithm 9) is sketched in Algorithm 11.

3.14 Implicit Time *k*d-trees

The implicit *k*d-tree's definition, construction and traversal given above in 3.5, 3.7, and 3.8 are hold general enough for also handling time as one dimension, such that four dimensional (3D + time) implicit *k*d-trees may be built above time varying scalar fields. Scientific visualization commonly displays such data by showing one time step after another. Implicit *k*d-trees built over four dimensions are not optimal here, as rays would also traverse time which is actually not needed, since for each rendering pass the time is predefined. Here it is smarter to use dim_t three dimensional implicit *k*d-trees (each requiring $dim_x \times dim_y \times dim_z$ memory) which have together the same memory requirements as one four dimensional implicit *k*d-tree (requiring $dim_x \times dim_y \times dim_z \times dim_t$ memory).

It is also possible to build one implicit *k*d-tree for all time steps by modifying the *get* function. In Algorithm 10 for example, Ω would have to be redefined:

$$\Omega = \cup_{i,j,k \in \{0,1\}, u \in \{0,\ldots,dim_t-1\}} s_{box[i][0],box[j][1],box[k][2],u}$$

This has the advantage that only one tree is stored for all time steps. The overall gain is not as major as one may initially expect, as still all scalar fields for all time steps have to reside in memory. For optimized implicit max *k*d-trees this only allows to visualize scenes less than 50% larger than before. This memory reduction scheme comes unfortunately for a rather high price: Nodes may be traversed even if their corresponding underlying scalar fields do not contain regions of interest at the currently predefined time. This may significantly reduce visualization speed, especially for strongly varying scalar fields. The low gains in memory savings does not justify the resulting major deceleration of visualization speed, such that this compression scheme is not further investigated here.

3.15 Test Environment, Test Scenario, and Test Data Sets

The computational results have been obtained using a dual dual-core 2.6 GHz AMD Opteron (64bit) desktop PC with 16 GB RAM as test environment. Tree construction and visualization (tree traversal) use all four processor cores.

For each scene two different traversal techniques have been used:

1. Single ray traversal (C++, single pixel)
2. Four ray traversal (SIMD, 2x2 pixels)

Tree construction- and their corresponding visualization-times for the different implicit *k*d-trees described above do not differ significantly. Therefore, the construction- and visualization times of optimized implicit max *k*d-trees are given here. They have a predefined memory overhead of one half the size of the underlying scalar field.

Following test setup has been used:

- the screen resolution is 512^2
- the scene is positioned with its center at the origin and once fully rotated around the vector (1, 1, 0) with 50 frames (7.2 degree rotation between each frame). Time varying data sets show for each new frame the next time step.
- The camera's vertical field of view angle is 45 degrees. It is positioned the bounding box' maximal diameter away from the origin on the z-axis, looking towards the origin.

The testing scenario has carefully been chosen, such that almost all rays of each frame enter the scene's bounding box and no high ray coherency is given. This assures that the evaluated frame rates are close to the worst case performances for the given scene types.

Following test data sets have been used for semi iso surfacing and for MIP visualization: bonsai(256^3, 1 byte) (Figure 3.4), stented abdominal ($512^2 \times 174$, 2 byte), seismic ($460 \times 567 \times 159$, 1 byte), beating heart ($512^2 \times 343$, 1 byte, 10 time steps) and two up sampled versions of chapel hill CT head (1024^3, 1 byte and 2048^3, 1 byte) (Figure 3.5).

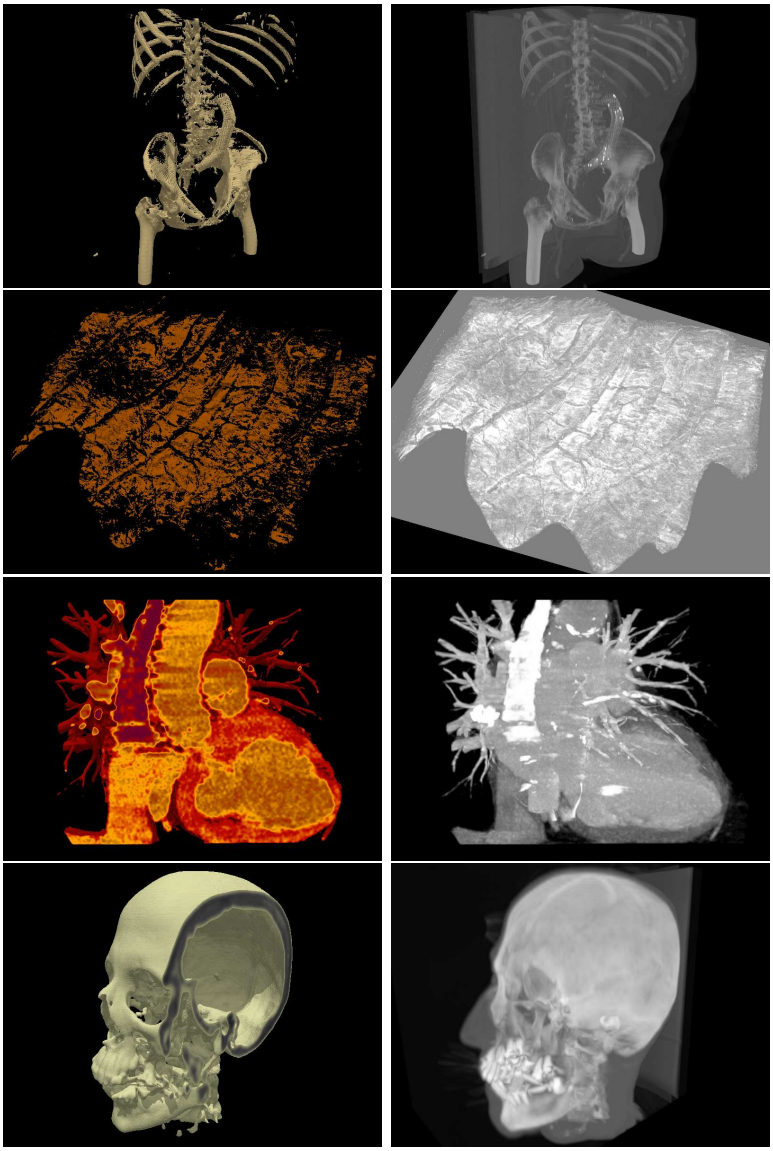

Figure 3.5: stented abdominal, seismic, beating heart and CT-head data sets. Note the slicing along the arbitrary near plane (Beating Heart) and the slicing along an axis aligned plane (CT-Head) with correctly cast shadows

The data sets have carefully been chosen, such that small (bonsai: 16MB), large (CT-head (large): 8GB), unequal scaled (stented abdominal: $dx = dy = 0.8398, dz = 3.2$) unequal sized (seismic: $dim_x = 460, dim_y = 567, dim_z = 159$), time varying (beating heart: 10 time steps), uniform (CT-head) and non-uniform (seismic) data sets are represented.

The 4D scalar field Beating Heart has been visualized by loading all time steps and all corresponding optimized implicit max *k*d-trees into main memory and visualizing them one by one.

3.16 Semi Iso Surfacing using Optimized Implicit Max *k*d-trees

For iso surfacing the intersection test and gradient calculation of rays entering leaf nodes has been implemented based on the technique described in (27). Phong shading with and without hard shadows is used.

Scene	iso + shadow		iso surfacing	
	C++	SIMD	C++	SIMD
bonsai	4.75	7.00	7.07	**12.5**
stent	6.21	9.58	7.80	**14.0**
seismic	4.32	6.28	5.51	**9.05**
beating heart	4.30	5.98	6.55	**10.7**
CT head	5.51	7.21	8.09	**12.6**
CT head (large)	4.75	5.98	6.99	**10.4**

Table 3.1: Frame rates (fps) for semi iso surfacing (with and without hard shadows)

The iso surfacing frame rates and SIMD-speedups given in Table 3.1 are even for the eight gigabyte data set interactive. The framerates are comparable to those achieved by the explicit (or there named "fat") *k*d-trees given in (50), while the here used optimized implicit max *k*d-trees require at most half as much memory as the "slim" *k*d-trees given in (50).

Note that the iso surfacing frame rates do not strongly depend on the scene size (due to the logarithmic dependency between visualization and scene size, as also experimentally verified in (50)). They are rather strongly affected by the scene type, as the comparison of seismic vs. CT-head (large) shows: The CT-head data set is relative uniform. Typically, after a few traversal steps the rays either hit the iso surface or leave the scene. On the contrary, the seismic data set is non-uniform: Even though seismic is much smaller than CT-head, it has lower frame rates due to its non-uniform character. Many rays repeatedly closely miss the iso surface until they finally leave the scene or hit the iso surface. This is equivalent to repeatedly traversing the tree up and down, which leads to a slower visualization.

The times needed for changing iso surfaces are not given, since the iso surface is changed as fast as new frames can be visualized (i.e. for all cases in less than 0.2 seconds).

3.17 MIP using Optimized Implicit Max *k*d-trees

For MIP it is possible to analytically calculate the maximum intensity of a ray inside a leaf node, but this is too expensive for interactive visualization. More common is to approximate the intensities, by sampling them along the ray's path using trilinear interpolation. The original implementation used here did sample the intensities at the rays' mid points of their enter- and exit-points of leaf nodes. But this leads to visualization artifacts, as maxima are often close to the leaf nodes' boundaries. Therefore SIMD trilinear interpolation at rays' exit points to the leaf nodes is used (similar as in (39)), which leads to nicer visualizations.

The MIP frame rates given in Table 3.2 are not as fast as for iso surfacing, which is due to the higher amount of tree traversal steps and interpolations. Interesting but not surprising is the SIMD speed up for different resolutions: As long as the leaf nodes' projections onto the viewport exceed the size of a single pixel, some SIMD speedup is observed. Once this is not given (as for the CT-head data sets) no sufficient ray coherency is given either, and the single ray traversal code outperforms the SIMD code.

The framerates for unmodified MIP are not for all scenes interactive. This is mainly due to the fact that no discretized grey levels have been used, such that visualization time is of order $O(n^{1/3})$. Therefore visualization with discretized grey levels (current maximum densities) has been implemented: If a ray's current maximum density is updated during tree traversal, it is discretized by fractioning it up to some given precision. Visualization time drops down to a mere order of $O(lg(n))$ (see (30)). Two discretization precisions have been used: 128 grey levels, with a visualization error of less than 1%, and 32 grey levels with an error of roughly 3%. The less precise visualization may be used during user interaction (rotation, translation, zoom). Once the user stops interacting, the picture with higher precision is visualized.

The implemented viewer allows to interactively switch between iso surfacing and MIP, where during MIP the iso surface slider is used to discard background noise: Only intensities above the value indicated by the iso surface slider will be visualized. This may strongly improve visualization speed. The framerates with discretized grey levels and the iso value slider set to the same value as for iso surfacing are given in Table 3.2.

Scene	non-discrete		discrete 128		discrete 32		32, no noise	
	C++	SIMD	C++	SIMD	C++	SIMD	C++	SIMD
bonsai	1.72	**3.02**	1.84	3.21	2.26	3.98	3.66	**6.48**
stent	1.39	**1.85**	1.70	2.33	2.48	3.59	4.98	**8.59**
seismic	0.68	**0.89**	2.23	2.74	2.42	2.99	4.12	**7.04**
beating heart	1.89	**2.38**	1.95	2.47	2.24	2.91	2.40	**3.11**
CT head	**0.73**	0.65	1.26	1.01	1.89	1.53	**3.38**	2.98
CT head (large)	**0.43**	0.31	0.90	0.53	1.52	0.81	**2.73**	1.65

Table 3.2: MIP frame rates (fps) for non-discrete, for 128, for 32 discrete grey levels, and for 32 discrete grey levels without background noise

As the new visualization order implies, the usage of discrete grey levels has especially for large scenes a strong positive effect on visualization time (see the CT-head scenes). Visualization speed also improved strongly for the Seismic data set, which may be related to its unstructured nature. Most important is that for

all data sets fairly interactive framerates are achieved, especially when discarding background noise.

All scenes except for the large CT head (8 GB) do not exceed one GB and have therefore also been visualized on a dual 3.2 GHz Intel Xeon (32 bit) desktop PC with 2GB RAM. The tree construction times and frame rates achieved on that machine are roughly half as fast as shown in Tables 3.1, 3.2, and 3.3 due to the half amount of processors given.

3.18 Parallel Tree Construction

Tree construction has been implemented iterative (using a stack for children nodes) and recursive. The recursive construction is used to create new threads, while the slightly faster iterative construction is used when only one processor core is left for the construction of the current subtree.

Scene	dimensions	construction times		
	(x × y × z)	1	2	4
bonsai	256 × 256 × 256	0.96	0.51	0.26
stent	512 × 512 × 174	2.85	1.50	0.78
seismic	460 × 567 × 159	2.13	1.09	0.56
heart (1 step)	512 × 512 × 343	5.19	2.73	1.38
CT head	1024 × 1024 × 1024	62.6	32.9	16.3
CT head (large)	2048 × 2048 × 2048	533	269	138

Table 3.3: Implicit *k*d-tree construction times (seconds) for 1, 2, and 4 processor cores

Tree construction times in Table 3.3 confirm the order of the construction algorithm. Construction times depend linearly on the scene volume and on the number of processor cores used.

Interesting is the overall time spent when using four processor cores, as here tree constructing takes roughly as much time as reading the entire uncompressed scalar field out of a non-cached hard drive region. It is therefore not reasonable to store implicit *k*d-trees separately on the hard drive, as there they would require

additional storage place, but reading this additional data would take roughly the same time as directly reconstructing the tree out of the underlying scalar field.

3.19 Future Research

Commonly the user is not only interested in iso surfacing and opaque slicing of the data set, but also in the semitransparent visualization of "interesting" regions. Those regions of interest are typically greater than (smaller than), or in between two iso values. Here, implicit *k*d-trees may also be used to accelerate the ray traversal of uninteresting regions where neither the trees, nor the ray-traversal algorithms need to be changed. Only the color calculation during ray-leaf intersection needs to be altered to change the iso surface viewer to a volume ray caster.

Figure 3.6: Stented Abdominal. Left: Iso surface representing bones and contrasted blood vessels. Right: Same scene, but visualized with a simple volume ray caster

Chapter 4

Interactive SIMD Ray Tracing for Large Deformable Tetrahedral Meshes

4.1 Abstract

A for general SIMD operations optimized ray tracer for large tetrahedral meshes that scales for static scenes sublinear to data size is presented here. In relation to other ray tracing systems this system achieves framerates that are comparable to the best render times in volume- and iso surface-rendering of those systems, where this system is additionally the most portable and by far the most general, such that it is best suited for portation upon different (future) hardware and for usage upon several applications.

4.2 Problem

Unstructured grids are widely used in numerical simulations to discretize the computational domain. They are for example popular in applications like computational fluid dynamics (CFD) and finite element methods (FEM). Although unstructured grids may contain a variety of different cell types (such as tetrahedra, hexahedra, or prisms), these grids can always be decomposed into a collection of

tetrahedra. The tetrahedral mesh is therefore the most important type amongst the unstructured grids.

Visualization of large tetrahedral meshes is a permanent challenge for computer science, and only visualization techniques that scale well to data size, that exploit the full potential of today's and future hardware, and which are very general are the once most likely to be used in the future.

Ray tracing is a candidate which fulfills all those requirements and additionally allows to produce high quality images. A glimpse at the near term goals of chip manufacturers reveals first that many core processors (i.e. processors with more than eight cores) are about to enter the market, and second that new processor cores will support up to 512 bit SIMD operations allowing sixteen single precision floating point operations at once. The inherently parallel nature of ray tracing makes it an ideal candidate for realization on many core architectures. But only ray tracers that use general SIMD operations to trace multiple rays at once are capable to exploit the full potential of today's and future hardware.

4.3 State of the Art

The amount of publications related to visualization of unstructured data is vast, and supplying a complete overview to all techniques would go beyond the scope of this chapter. Therefore the focus is here not on object order or hybrid techniques (i.e., techniques that use algorithms based on both image and object order techniques e.g., (6; 12)) but only on the image order technique ray tracing. For getting a more exhaustive overview to both image- and object-order rendering techniques for unstructured data it is referred to (26; 42)

Garrity (11) was the first to propose ray casting on convex tetrahedral meshes. Parker et. al (37) has been the first to achieve interactive framerates by using a supercomputer. The most noticeable ray tracing techniques that achieve interactive frame rates on today's desktop machines are the CPU-based rendering system proposed by Marmitt et al. (28) who used Plücker coordinates to traverse the tetrahedral mesh. Wald et al. (49) introduced a for 4-SIMD optimized coherent ray tracing system for visualizing tetrahedral iso surfaces. Weiler et

al. (54; 55) was the first to propose an interactive GPU-based fragment program ray caster.

4.4 Results

An interactive ray tracing method for large deformable tetrahedral meshes which has carefully been designed for supporting general single instruction multiple data (SIMD) operations is introduced here. It uses memory aligned and SIMD-friendly hierarchical acceleration structures and tetrahedron acceleration data allowing SIMD ray traversal and a newly proposed SIMD barycentric tetrahedron-plane intersection test. The method is very general and allows volume-, opaque- and accelerated (semi) iso surface-rendering as well as visualizing multidimensional data. It scales for static data sets sublinear to data size, such that it is well suited for visualizing massive data sets. The method does already achieve interactive framerates on today's multicore CPUs, where its design is general enough for portation on near future hardware which will be many core architectures supporting up to 512-bit register operations, allowing for each core to trace up to 16 rays at once through the mesh.

4.5 SIMD-friendly Acceleration Techniques

Besides the commonly known (SIMD-friendly) early ray termination and empty space skipping there exist several other acceleration techniques for ray tracing. The focus is only on SIMD-friendly techniques, where the acceleration techniques described in this section apply not only to tetrahedral but to general meshes.

4.5.1 The Memory Aligned Min/Max s*k*d-tree

As for all interactive ray tracing systems some acceleration structure in which the primitives are hierarchical allocated is needed. This hierarchical acceleration structure has to, as this is the prerequisite, support general SIMD operations. Four acceleration structures are commonly used for interactive raytracing: The

(hierarchical) grid and the *k*d-tree are spatial partitioning schemes while the s*k*d-tree and the BVH are object partitioning schemes.

Here are s*k*d-trees used, as its traversal routine is simpler than that for BVHs (two plane versus two bounding box intersections) and more SIMD friendly as the traversal of grids. The traversal routine is almost identical to that of the *k*d-tree, but the s*k*d-tree has a predefined maximal memory footprint, references each object only once, and uses commonly much less memory than the *k*d-tree or the BVH. The s*k*d-tree also supports deformable scenes (unlike spatial partitioning schemes that require a complete rebuild which is commonly slower), where it is updated in $O(n)$ time. The updates do not strongly degrade the s*k*d-tree's acceleration performance, as mesh deformations are typically not as severe as completely unstructured deformations.

4.5.1.1 Importance Bit

In order to allow to interactively toggle regions of interest on or of (Figure 4.1, top) are importance bits used. Each primitive ID's last bit is such an importance bit, specifying if the primitive is of interest. All tree nodes are also equipped with an importance bit, where a node is only marked as interesting if its sub-tree does contain at least one primitive of interest. Additionally, each leaf node contains a single all-important-bit specifying if all primitives inside the leaf are important or not (for its use see §4.5.2). The entire tree's importance bits are in a pre-processing step recursively set. The tree traversal routine and the primitive intersection test only enters tree nodes resp. tests primitives that are marked as important.

4.5.1.2 Leaf Bounding Box Minimization

During s*k*d-tree construction node splitting stops if two or less primitives are given within the current node. This minimizes the s*k*d-tree's maximal memory footprint to six nodes per primitive, but it increases the s*k*d-tree's disadvantage compared to BVHs, which is that its nodes' bounding boxes do not necessarily coincide with the bounding boxes around the primitives belonging to the nodes. This is an undesirable effect, leading to additional unnecessary ray primitive intersection

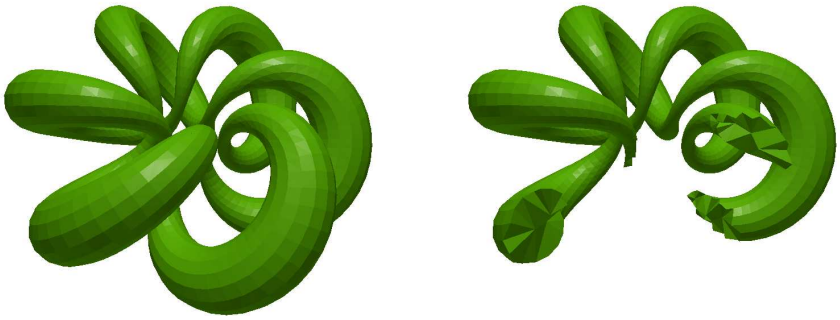

Figure 4.1: Data set shown once full, and once with deactivated primitives using the importance bit.

tests. Luckily, inside reasonable meshes this effect is not major, as the primitives are commonly closely packed within the mesh, such that the bounding boxes do not differ strongly.

Nevertheless, this effect exists and is not negligible especially at the mesh's boundary. Leaf nodes are therefore similar designed as the inner nodes. They do contain two clip planes, where each has its own orientation, direction and position (which stands in contrast to inner nodes, containing two split planes which have the same orientation and opposite directions).

After tree construction the leaf nodes are recursively minimized by placing the two clip planes at the two orientations and directions where the leaf bounding box differs most with the primitives bounding box (Figure 4.3). A leaf node is during tree traversal only entered, if some ray's near-far interval $[d_n, d_f]$ is still positive after the rays have been clipped against those clip planes (Figure 4.2).

4.5.1.3 Discretized Min/Max Values

As accelerated iso surfacing (see §4.5.3) shall also be supported, the s*k*d-tree is extended to a min/max s*k*d-tree, where during tree traversal a discretization of the interval $[s_{min}, s_{max}]$ representing the minimal/maximal scalar values given in the primitives belonging to the current node's sub-tree needs to be given (Figure 4.3).

The special property that the min/max of the current node coincides with some min/max of one of its child nodes (see also (14)) is used: Only inner tree

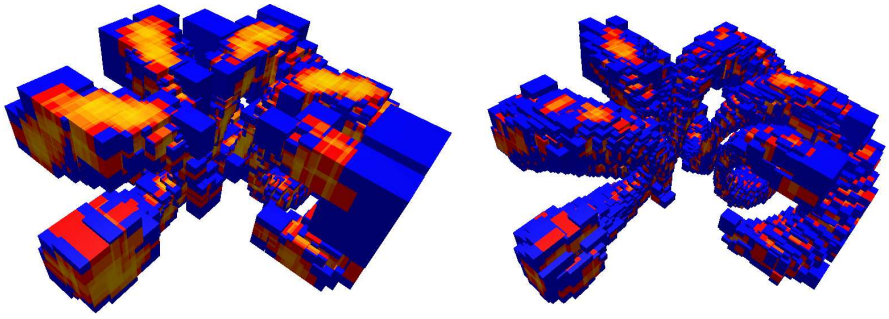

Figure 4.2: For the right scene in Figure 4.1 are the nearest hit leaf nodes displayed, once for non-minimized and once for minimized leaf bounding boxes. The different colors represent how many leaf bounding boxes touch this nearest hit point. Blue stands for one and red to yellow for two up to several leaf bounding boxes. Clearly leaf bounding box minimization adapts the leaf nodes better towards the scene helping to minimize the amount of ray primitive intersection tests.

nodes are equipped with discretized min/max values and two additional assignment bits. During traversal the min/max values of the current node are already known (the root node's min/max values are prior stored separately), the min/max values of the children node are initialized with the min/max value of the current node and then corrected by updating the correct min/max values of the children by using the two assignment bits.

Through this technique are during tree-traversal the min/max values of nodes already known in advance, such that only nodes whose min/max values specify them as interesting are touched. It is additionally not necessary to store the min/max values in leaf nodes, saving precious memory needed for storing other vital information such as the amount of primitives belonging to the leaf node and the clip planes' orientations and directions.

4.5.1.4 Efficient and Memory Aligned Storage Scheme

The node representation requires 32 bytes per node pair. The first node pair is an information node and the root node, where the information node stores the root node's min/max values and other information such as the total amount of

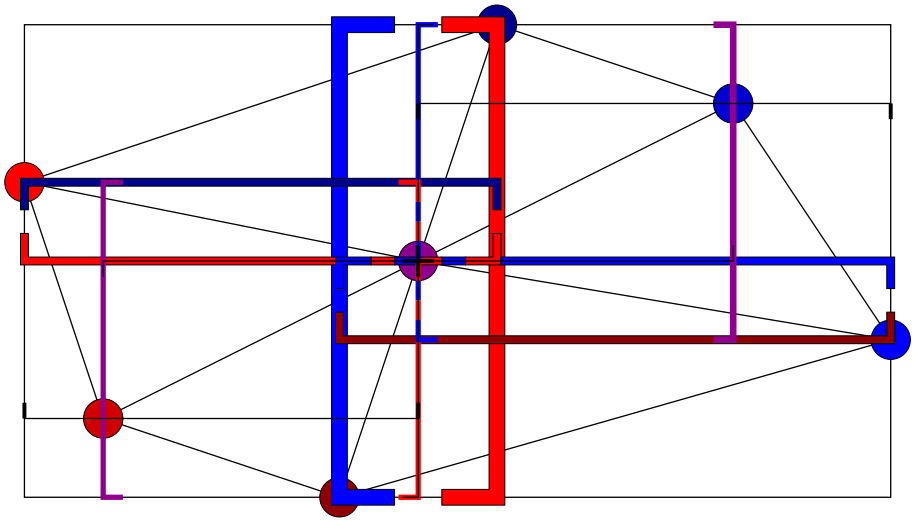

Figure 4.3: A min/max s*k*d-tree built over a small mesh. Each inner node has two split planes that are colored representing the minimal (blue) / maximal (red) scalar values given inside the node. Leaf nodes have up to two clip planes (thin black lines) that minimize the leaf's bounding box.

nodes. Each inner node has always two child nodes and contains therefore only one integer pointing to its both children. The entire tree is stored in a single 32-byte aligned array. Since the tree consists only of node pairs, it follows that all node pairs are stored with a cache friendly 32-byte memory alignment. A node's 16 bytes are used as given in Figure 4.4.

The first four bytes contain the information mask. Its first two bits define if the node is a leaf node, and if not, what orientation the inner node's split planes have. The next bit is the importance bit. In a leaf node are the next seven bit used for the all-important-bit and to specify the two clip planes' orientations and directions, the remaining 22 bits store the number of primitives belonging to the leaf node. In an inner node is the next bit empty and the remaining 28 bits are used for the two assignment bits and the discretized min/max values.

The next four bytes store some offset value, which is in an inner node the offset to its children nodes, while for a leaf node it is the offset pointing to the IDs of the primitives belonging to the leaf.

The last eight bytes store two floats, which are in an inner node the two split

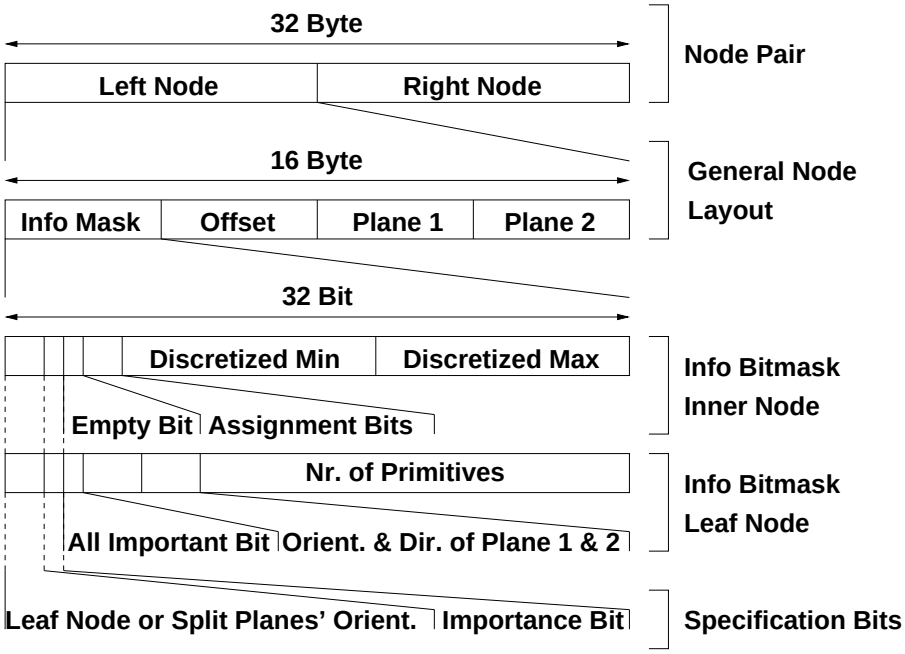

Figure 4.4: Memory layout of the skd-tree nodes

planes defining the left and the right child. In a leaf node they are the two clip planes minimizing the leaf's bounding box.

4.5.2 Memory Coherent Storage Scheme

SIMD ray tracing traces a bundle of rays at once and works best when the rays and the data are coherent, i.e., the ray bundle traverses similar regions and the rays are typically tested against neighboring primitives which are stored in the same memory region.

A mesh's primitives are often not stored in a memory coherent fashion, such that neighboring primitives may be stored at entirely different memory positions, leading to unstructured memory look ups when accessing the information belonging to nearby primitives.

The primitive information is therefore not stored in the order as the primitives have originally been sorted, but in a permuted form, which is defined by the underlying skd-tree. The primitive informations are in a preprocessing step ordered, such that those belonging to a node are all stored consecutively in memory. All

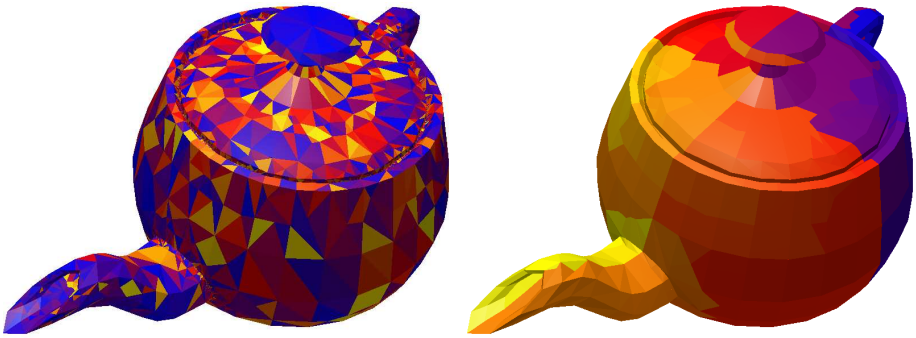

Figure 4.5: The primitives are colored according to their memory position, where blue over red to yellow indicate early to late memory positions. Left is the original primitive storage order given. Right are the primitives re-ordered using the underlying s*k*d-tree, which leads to a memory coherent storage scheme.

information belonging to one leaf node is therefore stored in one memory location, and adjacent leaf nodes commonly have nearby memory positions (Figure 4.5).

This minimizes memory look ups in leaf nodes typically to one, as leaf nodes that are marked as important do often contain only important primitives, which is specified by the all-important-bit. If this is the case it suffices to directly read the primitive information data without accessing the primitive IDs. Otherwise the memory block containing all primitive IDs is loaded first. The IDs' last bits specify if their corresponding primitives need to be tested for intersection.

4.5.3 Accelerated (Semi) Iso Surfacing

Iso surfacing corresponds to rendering the iso surface S_{iso} to the iso value s_{iso} in some scalar field S: $S_{iso} = \{s \in S : s = s_{iso}\}$ (Figure 4.6, bottom left).

Semi iso surfacing means not only to render the iso surface, but also values greater/smaller than the iso value $s \geqq s_{iso}$ / $s \leqq s_{iso}$. Semi iso surfacing is therefore a combination of iso surfacing and opaque rendering (in fact, rendering $s \begin{matrix}\leqq \\ \geqq\end{matrix} s_{iso}$ means rendering S which corresponds to pure opaque rendering, Figure 4.6 top).

Accelerated iso surfacing means ray tracing the implicitly given (and therefore not explicitly stored) iso surface by additionally skipping regions of no interest

(Figure 4.6, bottom right). The regions of interest depend on the iso value s_{iso} and the iso surfacing mode activated. For deciding if a region is of interest, the region has to be equipped with an interval $[s_{min}, s_{max}]$ that indicates the minimal/maximal scalar values given inside the region. A region is only of interest if $s_{min} \leqq s_{iso} \leqq s_{max}$ (iso surfacing) or $s_{max} \geqq s_{iso}$ / $s_{min} \leqq s_{iso}$ (semi iso surfacing).

Note that accelerated iso surfacing allows to change the iso surface on the fly, i.e., neither the iso surface nor the acceleration structure need explicitly to be re-calculated and stored. The implicitly given iso surface is directly displayed in the next rendering pass.

SIMD accelerated iso surfacing is almost identical to single ray accelerated iso surfacing. The only difference is that multiple rays skip regions of no interest or are tested against the implicitly given iso surface.

4.5.4 Incremental Traversal

Incremental traversal of meshes works by finding for a each ray the first hit cell and then traversing the mesh incremental by moving from the current cell to the next neighbor cell. Each face of a cell is therefore equipped with a primitive ID representing the corresponding neighbor cell.

SIMD incremental mesh traversal uses for each ray a separate current primitive ID, as different rays may enter and traverse different cells. For each traversal step, multiple cells are loaded, all rays in the SIMD packet traverse those different cells at once, and each ray's neighbor ID is updated to its next visited cell.

4.6 SIMD Ray Tetrahedron Intersection Tests

Unlike as in §4.5, this section's focus is explicitly on tetrahedral meshes by introducing new SIMD ray tetrahedron intersections tests.

The ray tetrahedron intersection test shall answer if a ray R defined by its origin R_O, direction R_D, and an interval defined by the near and far distance $[d_n, d_f]$, intersects a tetrahedron T which is defined by its four corner points T_i, $i = 0, 1, 2, 3$. If the ray tetrahedron intersection test turns out positive, it returns

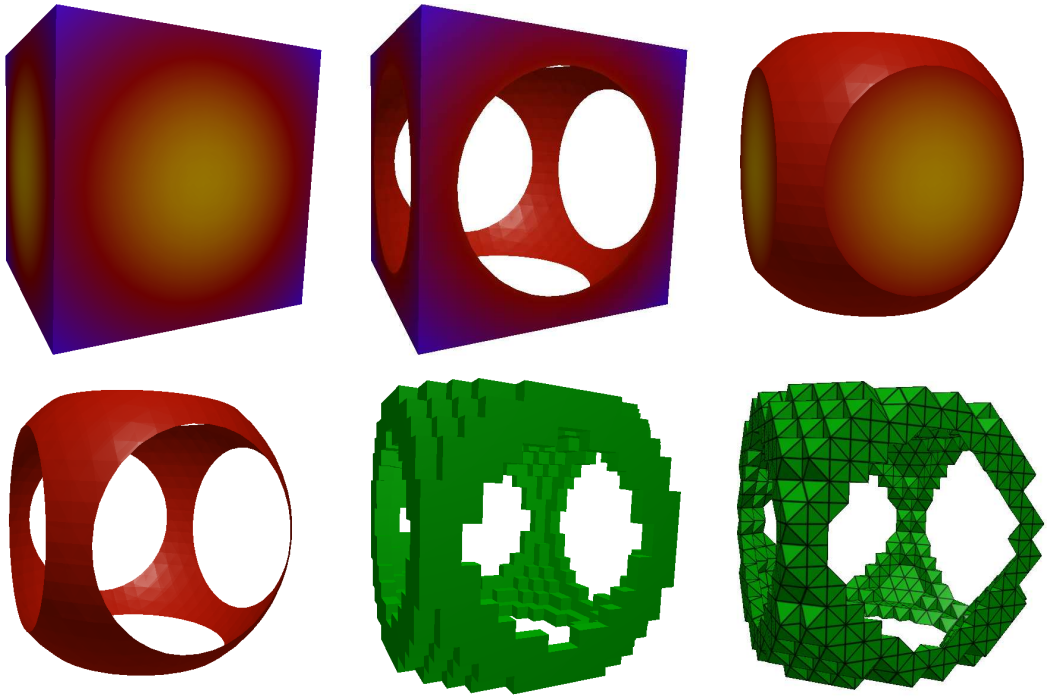

Figure 4.6: Data set visualized using opaque rendering, semi iso surfacing for values smaller/greater than the iso value, and iso surfacing. The last two images show for accelerated iso surfacing the leaf nodes of the s*k*d-tree and the primitives that are actually tested for intersection, since they contain parts of the iso surface.

the hit distance d_h together with the corresponding barycentric coordinates b_i. The test has also to be able to return the correct enter and exit point to the tetrahedron, which may not necessarily lie at the boundary of the tetrahedron (The rays' enter/exit point lies within the tetrahedron, if its near/far point $R_O + d_n R_D / R_O + d_f R_D$ lies within the tetrahedron (see Figure 4.7)).

4.6.1 Barycentric Tetrahedron-Plane Intersection Test

The above given requirements have to be taken into account when implementing a SIMD friendly ray tetrahedron intersection test. Especially the SIMD friendliness is quite demanding, as the test may return completely different results for its different rays in the SIMD ray packet. Testing for example each ray against all four triangle faces of the tetrahedron turns out to be non SIMD friendly,

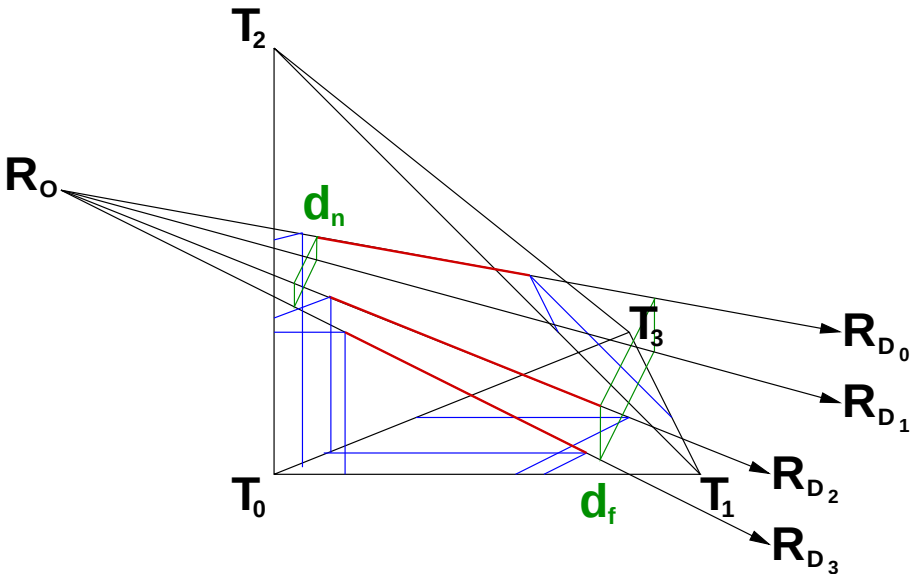

Figure 4.7: Some of the multiple rays tested may either enter/exit the tetrahedron at different faces or inside the tetrahedron, or even entirely miss the tetrahedron

as different rays may intersect different triangles or may completely miss the tetrahedron.

Instead, each ray's interval $[d_n, d_f]$ is updated by testing each ray against each of the four planes P_i defined by the three points $\{T_{i\hat{+}1}, T_{i\hat{+}2}, T_{i\hat{+}3}\}$, where $i\hat{+}j \mathrel{\widehat{=}} (i+j) \bmod 4$. The orientation of the plane defines if either the near d_n or the far d_f value is candidate for update. The candidate will only be updated to the ray plane intersection distance d_P, if this distance is greater than d_n resp. smaller than d_f. After the rays have been tested against the four planes, the simple comparison $d_n < d_f$ answers for each ray, if the ray hits the tetrahedron or not (see the red marked intervals in Figure 4.7).

A standard ray plane intersection test to a plane P uses the normalized plane normal N and a scalar value o that is equal to the minimal distance of P to the origin. Evaluating for some given d the equation

$$(R_O + dR_D)N - o = x \qquad (1)$$

returns the distance of the point $R_O + dR_D$ to the plane P. Testing R against P means finding the plane hit distance d_P that solves this equation for $x = 0$

$$d_P = \frac{o - R_O N}{R_D N}$$

But the interest is not in the distance to the plane P but in the barycentric coordinate b, such that not a normalized N (as for example proposed in (54)) is used. Instead, some modified N_i and o_i are used (Figure 4.8). They are evaluated by first calculating the scalar values v_i.

$$v_i = (T_{i\hat{+}2} - T_{i\hat{+}1}) \times (T_{i\hat{+}3} - T_{i\hat{+}1}) \cdot (T_i - T_{i\hat{+}1})$$

$$N_i = \frac{1}{v_i}(T_{i\hat{+}2} - T_{i\hat{+}1}) \times (T_{i\hat{+}3} - T_{i\hat{+}1}), \qquad o_i = N_i \cdot T_{i\hat{+}1}$$

The geometric interpretation to solving equation 1 with these N_i and o_i is that for each d the volume of the tetrahedron spanned by the four points $\{T_{i\hat{+}1}, T_{i\hat{+}2}, T_{i\hat{+}3}, R_O + dR_D\}$ divided by the volume of the tetrahedron T is evaluated. This is equal the barycentric coordinate b_i.

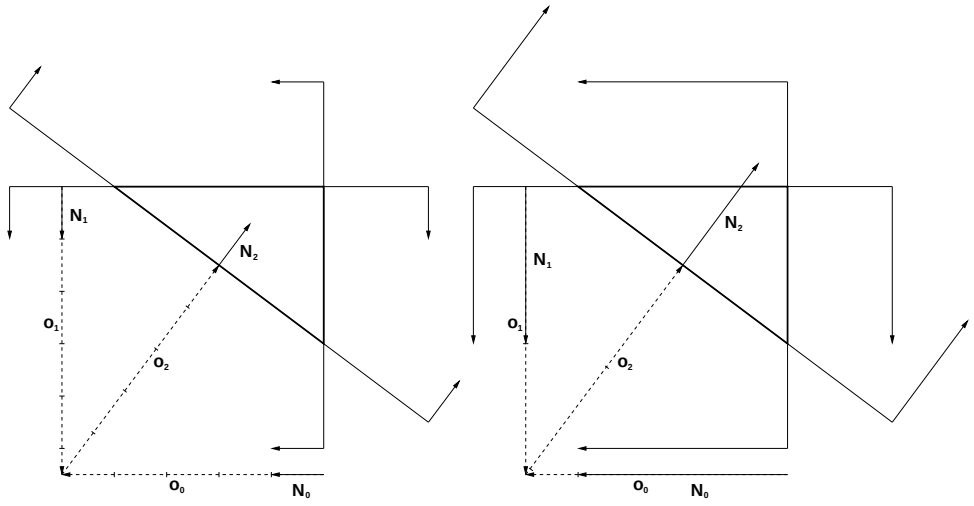

Figure 4.8: 2D example for Normals N_i and offsets o_i for the barycentric tetrahedron-plane intersection test. Left: Normalized normals with the corresponding offsets. Right: Spezial normals and offsets for which the ray plane intersection test directly returns a distance that equals the corresponding barycentric coordinate.

4.6.2 Memory Aligned Tetrahedron Acceleration Data

Minimization of memory look ups is of high interest when keeping many core architectures in mind, as memory bandwidth will most likely be the major bottleneck to be overcome. Therefore a 96 byte large tetrahedron acceleration data containing all the necessary information in pre-computed form is proposed here (Figure 4.9).

$N_{0,x}$	$N_{1,x}$	$N_{2,x}$	$N_{3,x}$
$N_{0,y}$	$N_{1,y}$	$N_{2,y}$	$N_{3,y}$
$N_{0,z}$	$N_{1,z}$	$N_{2,z}$	$N_{3,z}$
o_0	o_1	o_2	o_3
s_0	s_1	s_2	s_3
n_0	n_1	n_2	n_3

Figure 4.9: Memory layout of the tetrahedron acceleration data.

The acceleration data allows fast ray tetrahedron intersection tests, where one 32 byte aligned memory look up suffices to receive all data required. The entire data is independent of any ray R and is therefore only computed once in a pre-processing step.

A standard ray tetrahedron intersection test for tetrahedral meshes needs at least five memory look ups (one for the vertices' IDs, four for the vertices' data). To minimize the memory look ups to one, it suffices to store for each tetrahedron the corresponding vertices data in a separate array. In order to keep the data memory aligned 64 bytes are used to store the N_i and o_i directly, which

also saves costly computations for the ray tetrahedron intersection test. Using this acceleration data, the ray tetrahedron intersection test simplifies down to one aligned memory look up and four barycentric tetrahedron-plane intersection tests, where each of these tests needs only two scalar products, a division and two conditional operations (plus an addition and some comparison operations). The barycentric coordinates at the hit point are then calculated by re-using the scalar products that have prior been evaluated (Algorithm 12).

The tetrahedra's neighbor IDs are stored in the next 16 bytes, where the last bit of each neighbor ID specifies if the corresponding tetrahedron is toggled on/off.

The last 16 bytes are used to store the scalars s_i belonging to the tetrahedron's four vertices. This stands in contrast to (54), where 16 bytes are used to store the constant gradient g and some offset scalar which are then used for directly evaluating scalar values at a ray's point. This scheme is not followed here, as it discards the possibility of applying a fast min/max check that is needed for accelerated iso surfacing. Nevertheless it is possible to inexpensively calculate the scalar values along the rays, as the barycentric tetrahedron-plane intersection test evaluates the barycentric coordinates on the fly, which are then used to evaluate the scalar values at the hit points.

Note that the N_i that are stored inside the acceleration data coincide with the directions of the four tetrahedron faces. Furthermore may the gradient g be won by simply evaluating $g = (N_0|N_1|N_2|N_3|)(s_0, s_1, s_2, s_3)^T$. The five normals of interest (belonging to the tetrahedron's four boundary faces/the iso surface) may therefore be won by simply normalizing N_i/g. This normalization is only needed for shading purposes and is typically not expensive, since SIMD instruction sets commonly do supply a fast approximation of the inverse square root.

4.6.3 Basic Sample Code

The (SIMD) ray tetrahedron intersection tests are based on the (single ray) C++ sample code supplied in Algorithm 12

This single ray intersection test may actually be converted into two different types of SIMD ray intersection tests.

Algorithm 12 Ray tetrahedron intersection test

```
unsigned int intersectTetrahedron(float *rayO, float *rayD, float &dN,
                                  float &dF, unsigned int id, float *b)
{
  float *a = accel + 24 * id, s[4], d[4], rO[4], rD[4], n = dN, f = dF;
  for(int i = 0; i < 4; i++)
  {
    rO[i] = a[i] * rayO[0] + a[4 + i] * rayO[1] + a[8 + i] * rayO[2];
    rD[i] = a[i] * rayD[0] + a[4 + i] * rayD[1] + a[8 + i] * rayD[2];
    d[i]  = (a[12 + i] + rO[i]) / rD[i];
    s[i]  = rD[i] > 0.0f;
    n     = (s[i] && n <= d[i])? d[i] : n;
    f     = (s[i] || f <  d[i])? f : d[i];
  }
  if(n >= f)
    return 0;
  dF = n;
  for(int i = 0; i < 4; i++)
    b[i] = rO[i] + n * rD[i] - a[12 + i];
  return 1;
}
```

One possibility is to load one tetrahedron and to test one ray against the four planes at once. Here the updates of the near and far values `n` and `f` have to be extended by some non-SIMD-friendly horizontal maximum and minimum operation.

The other possibility is to load one or multiple tetrahedra and to test multiple rays against it resp. them. Here the statement `(n >= f)` may differ for different rays, such that the `if` statement is inappropriate and has to be replaced by some other control flow.

The first possibility has the major drawback that it is only suited for SIMD operating on exactly four elements. As the goal is a ray tetrahedron intersection test for general SIMD operations the focus on the second possibility.

To cope with the problem of different results for different rays a help variable `r` is introduced. It is used for applying conditional operations at the statements

after the position of the removed `if` statement.

Algorithm 13 Ray tetrahedron intersection test: SIMD friendly modification

```
r    = (n >= f);
dF   = r? dF : n;
for(int i = 0; i < 4; i++)
  b[i] = r? b[i] : rO[i] + n * rD[i] - a[12 + i];
return r;
```

This modification is fully SIMD friendly and returns the same results as the above given code sample.

4.6.4 Implementation of Different Interesection Tests

Different visualization modes (opaque-, iso surface- or volume-rendering) require different ray tetrahedron intersection tests and use modified or extended versions of the above given ray tetrahedron intersection test. The SIMD ray tetrahedron intersection tests have to return for each ray the distance and the barycentric coordinates to the...

- first hit Tetrahedron (opaque rendering)
- iso surface intersection (iso surface rendering)
- exit point of the current tetrahedron (volume rendering)

Each visualization mode has also further special requirements to its corresponding intersection test.

4.6.4.1 Opaque Rendering

Opaque rendering works by testing multiple rays against single tetrahedra, where either one of the four surface normals of the hit tetrahedron, or - if the ray's near point lies within the tetrahedron - the normal of the clipping plane that caused the ray to start at this near point has to be returned. For assigning the normals correctly, it is crucial that the barycentric coordinate of an on the corresponding boundary face hit tetrahedron equals zero. The barycentric coordinate of the

hit plane is therefore explicitly set to zero, as this coordinate may be - due to numerical instabilities - unequal zero.

4.6.4.2 (Semi) Iso Surface Rendering

Iso surface rendering also works by testing multiple rays against single tetrahedra, where the ray tetrahedron intersection test is extended by the initial check if the tetrahedron contains regions of interest (and if not, the test returns directly "no hit"). For iso surfacing the scalar values at the enter and the exit point to the tetrahedron are evaluated. If both values are smaller or greater than the given iso value the test returns directly "no hit". Otherwise the iso surface intersection point is evaluated by linear interpolation, and the corresponding normal (which is the normalized positive/negative gradient in the tetrahedron) is returned.

For semi iso surfacing, after the evaluation of the rays' enter values, it is additionally checked if the enter values are greater/smaller than the iso value. If so, the test is terminated, and the same normal as for opaque rendering is returned.

4.6.4.3 Volume Rendering

Volume rendering works by testing multiple rays against multiple tetrahedra, where each ray is traversed from one tetrahedron to the next neighboring tetrahedron. For each ray the first hit tetrahedron is therefore needed, which is evaluated by using the opaque rendering mode as initialization. As different rays may traverse different tetrahedra, each ray is equipped with an own tetrahedron ID. Multiple tetrahedra acceleration data are loaded, converted (using SIMD operations) into SIMD-friendly data layouts, and the scalar values at the far points as well as the IDs to the next tetrahedra are evaluated and returned.

4.7 Different Visualization Features

The in §4.6 introduced ray tetrahedron intersection tests have, besides the major advantages of being SIMD- and cache-friendly, the additional benefit of always returning the correct intersection points and barycentric coordinates, even if the

rays start and/or end inside the tetrahedra. Two direct gains from this benefit are that the data set has neither to be convexificated (as in (54)), nor need its boundary triangles be known (and therefore stored) explicitly for initializing the tetrahedral mesh traversal (28; 54). Even more remarkable are the many new features that evolve out of this benefit.

4.7.1 Mesh Display Techniques

The first features work directly on the tetrahedral mesh and are independent of the data values stored inside the mesh. They are all independent of each other and may be activated simultaneously.

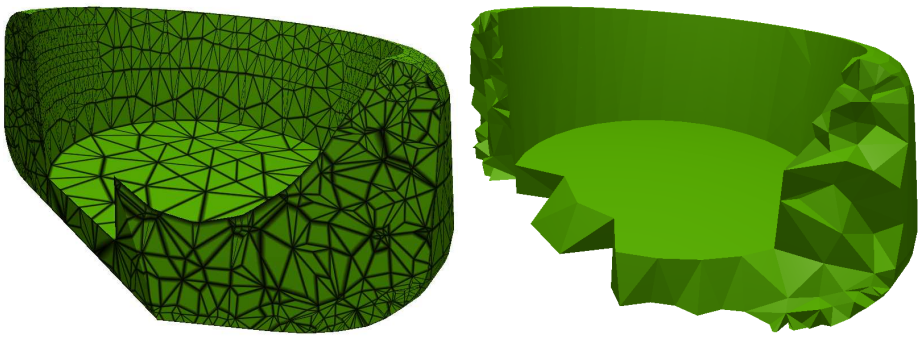

Figure 4.10: Dataset clipped along an axis aligned plane and the arbitrary near plane. Left are the mesh's edges displayed, revealing the mesh's structure at its boundary. Right is mesh clipping activated, giving insights to the mesh's inner structure.

4.7.1.1 Arbitrary Plane Clipping

A direct gain is that it is possible to interactively slice the tetrahedral mesh along arbitrary planes. The data set is automatically clipped against the arbitrarily positioned view frustum near plane and additionally - as in any case this has to be performed prior to traversing a skd-tree - against some axis aligned box.

4.7.1.2 Mesh Clipping

Instead of clipping the mesh along smooth planes it is also possible to clip the mesh along its primitives - in this case the tetrahedra. A tetrahedron will only be tested for intersection if all of its four vertices lie within the the axis aligned bounding box and behind the near plane. As opaque rendering does return correct normals, mesh clipping gives clear insights how the mesh is internally structured.

4.7.1.3 Mesh Edges

Even though mesh clipping is perfectly suited for displaying inner grid cells, it is not ideally suited to display the mesh's quality at its boundary. The tetrahedral mesh's edges are therefore displayed by adaptively blending the edge color above the underlying color.

4.7.2 Visualizing Multidimensional Data

The here described features operate on the data values stored inside the mesh. They all use each ray's barycentric coordinates of the hitpoint to interpolate the values given on the hit tetrahedron's vertices. All features are independent of each other and may be used simultaneously (Figure 4.11), which allows to directly visualize multidimensional data by applying different result values to different features. All these techniques may additionally be combined with the different mesh display techniques described above.

- **Iso surfacing** works by only traversing the min/max s*k*d-tree nodes that do contain regions of interest (see §4.5.3). If a leaf node containing regions of interest is traversed, all rays are tested against the tetrahedra contained in the leaf as described in §4.6.4.2.
- **Iso lines** are visualized similar as the mesh's edges. If a ray's scalar value at its hitpoint differs less than a given threshold from an iso-line value, the line-color is adaptively blended above the hit point color.
- **Coloring** works by using a ray's scalar value at the hit point for reading the resulting color out of a color table.

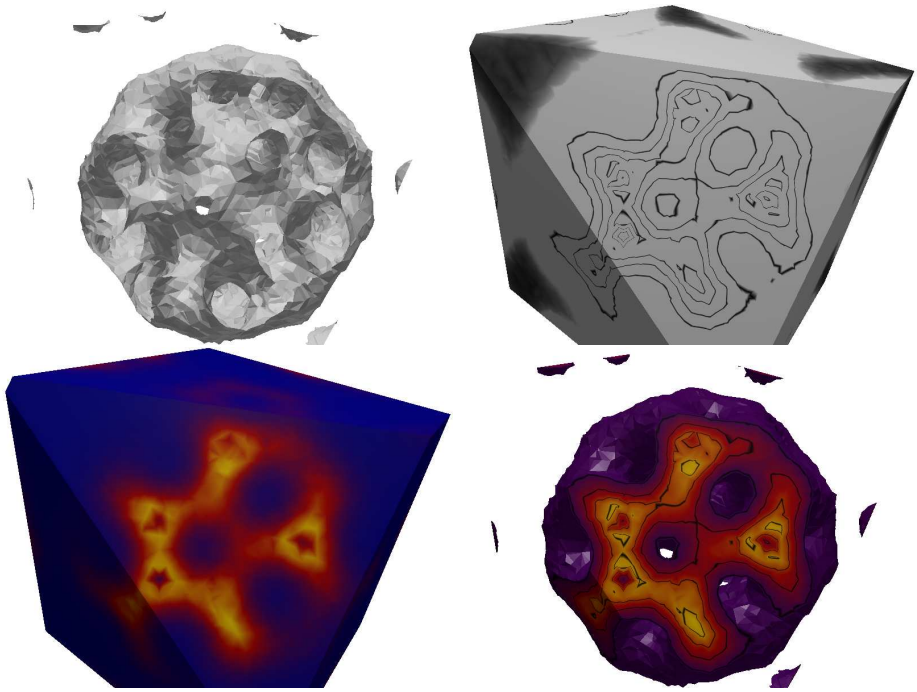

Figure 4.11: A dataset clipped along an axis aligned plane and the arbitrary near plane is visualized using iso surfacing, iso lines, and coloring. In the last image is semi iso surfacing used, which allows to combine all techniques, revealing more information than in any of the other three images.

4.7.3 Volume Rendering

Volume rendering (Figure 4.12) works by traversing tetrahedra as described in §4.6.4.3, where for each ray the corresponding opacity and color is updated after each tetrahedron traversed. Pre-integrated volume rendering is not applied, as it is not SIMD-friendly. Instead a simple one dimensional color table is used for looking the results up, as the color table itself is small enough to reside in cache. This adds some little computational overhead in comparison to pre-integrated volume rendering but discards the cache-unfriendly table look ups.

SIMD volume ray tracing of non-convex data sets is efficiently implemented by simply starting new rendering passes for ray packets containing rays which did traverse parts of the volume, where the new rendering pass is initialized with the rays' exit points to the volume (Figure 4.13).

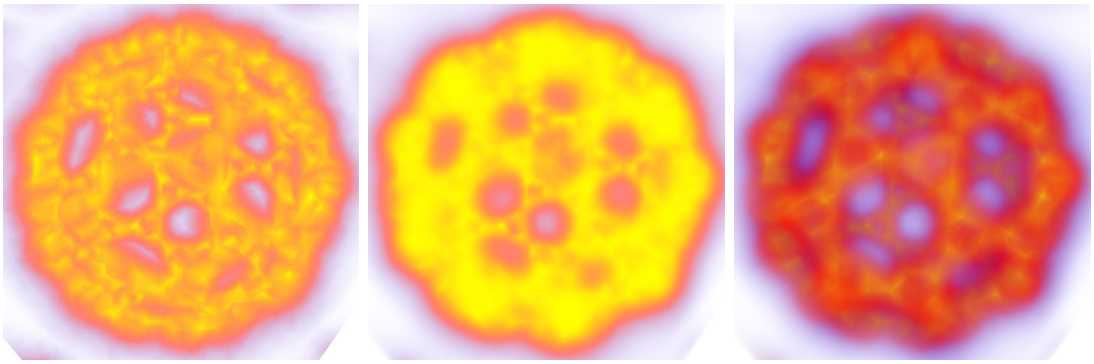

Figure 4.12: The same data set visualized by three rendering modes: maximum intensity projection, xray, and volume rendering using an emission-absorption model, where the same transfer function is applied to each render mode.

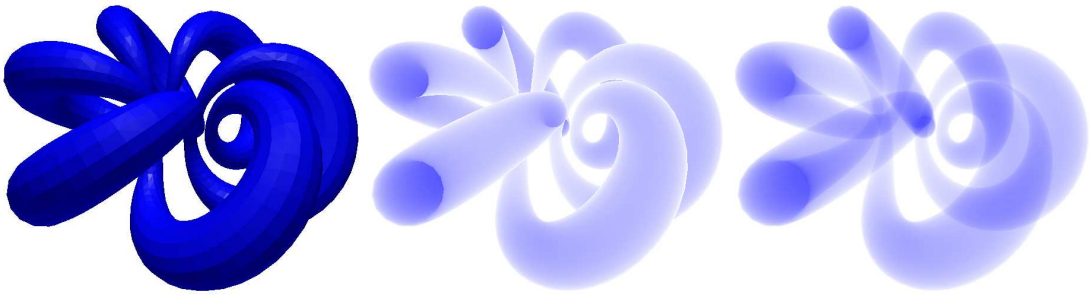

Figure 4.13: The left data set is volume rendered twice using an emission-absorbtion model, first without and then with support for non-convex data sets.

4.8 Results and Discussion

4.8.1 Scalability

To show, how the SIMD ray tracing method scales when changing

- to a single-ray ray tracer
- the viewport size
- the data size
- the amount of processor cores

artificial datasets have been created by partitioning the unit cube into $(2^q)^3$ sub-cubes, where each sub-cube has been decomposed into 5 tetrahedra resulting in $n(q) = 5 \times 8^q$ tetrahedra. The scalar values on the vertices have been set

proportional to their distance to the origin. Quadratic viewports have been used, where the viewport side lengths where set to 2^v, resulting in $p(v) = 4^v$ pixels. The camera has been positioned the unit cube's diameter's length away from the origin on the z-axis focusing with an 45° angle of view towards the origin (see Figure 4.14). Iso surfacing has been executed with the scalar value set to the maximal value where the corresponding iso surface did still represent an intact sphere. Volume rendering has been executed with deactivated early ray termination, such that all rays traversed the entire volume. The speed up and the average render times in seconds per frame have been evaluated by rotating the test data sets once fully around the vector $(1, 1, 0)$ using opaque-, iso surface- and volume-rendering. If not stated differently $q = 6$ and $v = 9$ have been used on a 2 GHz Core1 Duo (which corresponds to 1.31 Mio. tetrahedra, a 512^2 viewport and $c = 2$ processor cores). The results are given in Figure 4.15.

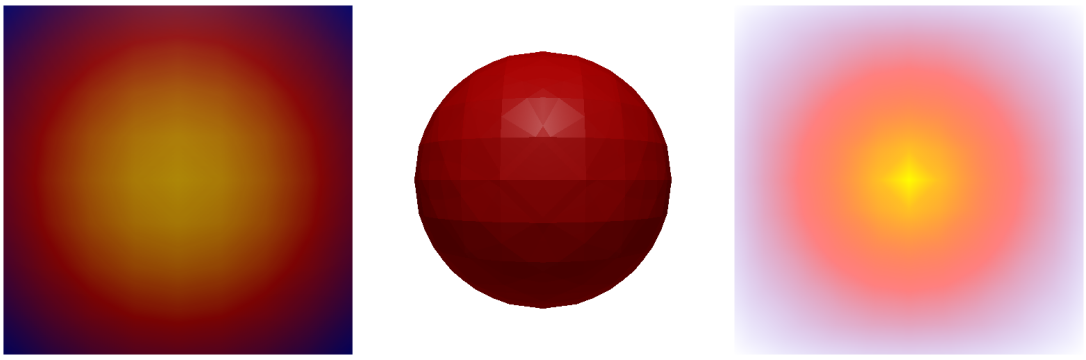

Figure 4.14: First frame for a test data set (here for q = 3) rendered opaque, iso surfaced and volume rendered.

Interesting is the SIMD speed up for different data sizes:

For iso surface and for opaque rendering the SIMD speed up decreases gradually when increasing the data size, which is explained through the decreasing data coherency. Nevertheless speed ups between 2.3 and 1.6 are still achieved. If the viewport is increased the speedup will also improve.

For volume rendering it is observed that the speed up is at the beginning at about 1.9 and stabilizes then at about 1.7. This is explained by the different technique volume rendering uses. It loads for each ray the corresponding tetrahedron acceleration data, where the entire data has to be reordered into a

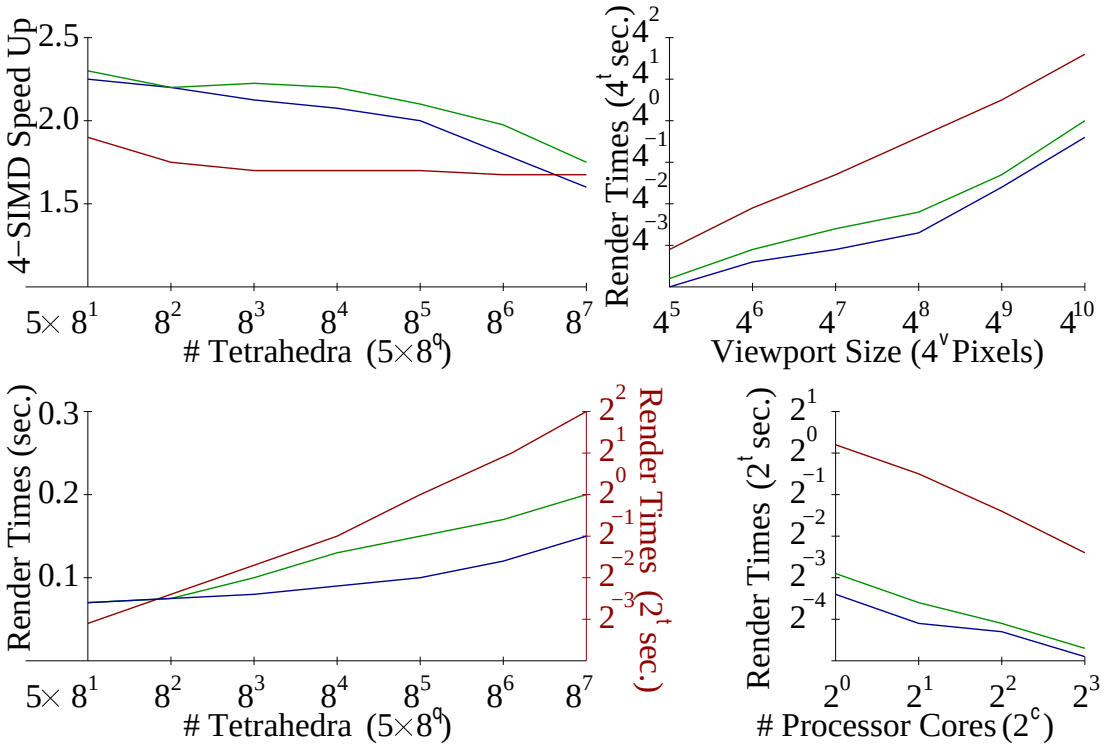

Figure 4.15: Result graphs for opaque, iso surface, and volume Rendering. Upper left: Speed up of the 4-SIMD ray tracer compared with the corresponding single-ray ray tracer for $q = 1 \ldots 7$. Upper right: Render times to viewport size for $v = 5 \ldots 10$. Lower left: Render times to data size for $q = 1 \ldots 7$, where iso surface and opaque rendering are plotted on a lin/$\log_8$ scale, while volume rendering is plotted on a $\log_2/\log_8$ scale. Lower right: Render times on $c = 2^0 \ldots 2^3$ 2.5 GHz Xeon processors.

SIMD-friendly storage scheme. This additional setup operation decreases speed up. But, unlike as for the other two render modes, volume rendering does not suffer the problem of decreasing coherency with increasing data size, such that it's speed up stays constant. The higher speed up for small data sets is explained through the fact that the volume rendering time is for small data sets dominated by the initialization time (opaque rendering), which itself has a higher speed up.

4.8.2 Complexity Analysis

In the complexity analysis the focus on the complexity of setup- update- and render-time with respect to the amount of tetrahedra n. The render time t does of course not only depend on n but also on the amount of pixels p and processor cores c used. But this dependency is basically linear (as also experimental verified in the graphs at the right of Figure 4.15) such that the focus is only on $t(n) \approx t(n,p,c)c/p$.

4.8.2.1 Setup Time, Update Time

The setup is done only once when loading the data set. Its time is of order $\mathcal{O}(n\ lg(n))$ as it is dominated by the construction time of the skd-tree $\mathcal{O}(n\ lg(n))$ and tetrahedron acceleration data $\mathcal{O}(n)$.

Updates are of order $\mathcal{O}(n)$, and are performed after deforming the scene or changing the regions of interest which both require updating the skd-tree $\mathcal{O}(n)$ and tetrahedron acceleration data $\mathcal{O}(n)$.

4.8.2.2 Render Time: Worst Case Scenarios

In worst case scenarios, brute force visualization of n opaque objects takes $\mathcal{O}(n)$ operations. If the objects are semi-transparent, and some non-commutative blend function shall be applied correctly, the objects have to be additionally sorted, which increases the amount of operations to $\mathcal{O}(n\ lg(n))$. This holds for both image order and object order techniques.

For tetrahedral meshes, the order for opaque (iso surface) ray tracing by using an underlying (min/max) skd-tree does actually not change for worst case scenarios. For volume ray tracing the order improves to $\mathcal{O}(n)$, when using incremental traversal upon a convex tetrahedral mesh.

4.8.2.3 Render Time: Real Life Examples

But this holds only for artificial constructed worst case scenarios. For reasonable tetrahedral meshes the order for opaque (accelerated iso surface) ray tracing by using an underlying (min/max) skd-tree is rather of order $\mathcal{O}(lg(n))$, while volume

ray tracing (standard iso surfacing) using incremental traversal combined with an skd-tree (for finding the first hit tetrahedron) is rather of order $\mathcal{O}(lg(n)+n^{1/3}) = \mathcal{O}(n^{1/3})$.

This theoretical observation is also experimentally verified in the lower left graph of Figure 4.15, where the graph scales are chosen in such a manner that ideal results would represent straight lines.

4.8.2.4 Comparison to Hybrid and Object Order Techniques

Hybrid and object order techniques both iterate at least once over all primitives and are therefore at least of linear order $\mathcal{O}(n)$ or even worse of order $\mathcal{O}(n\ lg(n))$. This stands in contrast to ray tracing which is for static data sets of sublinear order $\mathcal{O}(lg(n))$ (opaque or accelerated (semi) iso surface rendering) or $\mathcal{O}(n^{1/3})$ (volume rendering or standard iso surfacing). For static data sets ray tracing does therefore outperform object order and hybrid techniques at some given data size.

4.8.3 Memory Requirements

As the ray tracing system scales sublinear to data size, it is perfectly suited to visualize large data sets, for which the memory requirements are of high interest. Therefore the memory requirements are provided here, where both the fixed and the variable memory requirements (which are independent resp. dependant of the underlying tetrahedral mesh) are given. For the variable memory requirements is the typical size and the theoretical maximum given (Table 4.1).

	Fixed	Variable Typically	Variable Max
Mesh	16	(0.2 to 0.25)$(12+4s)$	$4(12+4s)$
skd-tree	4	16 to 32	96
Acceleration	96	0	0

Table 4.1: Memory requirements in bytes per tetrahedron, where s stands for the amount of different scalar values (the dimensionality of the data) assigned to the vertices

The theoretical maximum of the memory requirements is $260+16s$ bytes, but this

is an upper bound which is only reached by artificial constructed data sets (e.g., a random set of non-connected tetrahedra). All data sets used in this chapter (and typically all non-small and non-artificial constructed data sets) do not exceed a memory requirement of 150 bytes per tetrahedron.

The system does not need any additional memory during setup, as first an array of 96 bytes per tetrahedron is allocated, into which the s*k*d-tree is built. The s*k*d-tree's size (which is typically much smaller than the theoretical upper bound) is known after this build, such that this array is reallocated to fit the s*k*d-tree's size. Then a second array with 96 bytes per tetrahedron is allocated. It is first used to hold temporary data required for evaluating and storing the tetrahedra's neighbors. After this evaluation the array is then used to store the neighbors and the remaining parts belonging to the tetrahedron acceleration data.

4.8.4 Speed, Correctness, Portability and Generality

Visualization systems do often trade speed versus correctness versus portability versus generality, e.g., a system is fast, but it does not display the results correctly and/or it is bound to specialized hardware and/or it lacks the ability to offer many features.

The system described here has been designed for meeting all four requirements: It achieves fast render times, it displays the data correctly (as it uses the high quality technique ray tracing, which also displays for example topological cycles correctly), it is not restricted to specialized hardware and it is very general.

For getting an impression how portable/general the SIMD ray tracing technique is, a comparison to the three other current state of the art ray tracing techniques for tetrahedral meshes is provided, where each feature is rated as

☑ implemented

☐ not yet implemented, but possible without major changes

☒ only with major changes or not at all implementable

Table 4.2 shows that the system described here is the most portable and by far the most general. This is mainly achieved through the newly introduced SIMD barycentric tetrahedron-plane intersection test which always returns correct results, through the fact that the render modes are initialized by finding the initial

Feature	(49)	(28)	(54)	Here
General SIMD compatibility	☒	☒	☐	☑
Not Restricted to Special Hardware	☑	☑	☒	☑
Iso Surfacing	☑	☑	☑	☑
Accelerated Isosurfacing	☑	☒	☒	☑
Semi Iso Surfacing/Opaque Rendering	☒	☐	☐	☑
Volume Rendering	☒	☑	☑	☑
Slicing	☑	☒	☒	☑
Multidimensional Data	☒	☐	☒	☑
Deformable Scenes	☑	☐	☐	☑
Toggle Regions of Interest on/off	☐	☒	☒	☑
Shadows	☑	☑	☐	☑

Table 4.2: portability (top) and generality (bottom) in terms of features for the three state of the art ray tracers compared with the here described ray tracer.

hit tetrahedron and not triangle which allows to interactively slice through the data set or to toggle regions of interest on/off, and through the min/max s*k*d-tree which allows accelerated iso surfacing/toggling regions of interest on or off.

4.8.5 Comparison to other Ray Tracing Systems

When comparing this ray tracer to the three ray tracers (28; 49; 54), it has to be distinguished between volume rendering and iso surfacing.

(54) (and (55)) report volume rendering framerates that are comparable to those resulting when running the here described system on two CPU-cores. That system is bound to the small on board memory of the graphics card and is therefore not capable of visualizing as large data sets as the here described system does (as it may use the machine's main memory).

(28) uses the simple boundary triangle intersection for volume rendering initialization and does only test necessary tetrahedron faces against the rays. Those techniques help to speed up the ray tracing process, where they have the disadvantages of lower generality and being SIMD-unfriendly. The here described system is more general and achieves through its SIMD-friendliness comparable

frame rates.

(49) has exclusively been built for accelerated iso surfacing using aggressive packet and frustum traversal, such that it achieves compared to the here described system faster/slower framerates for small/large sized data sets respectively (as small/large data sets have high/low data coherency which effects frustum traversal more positive/negative than SIMD traversal). That system is the least general, as it is not based on a ray tetrahedron but on a ray iso polygon intersection test and can therefore neither support multidimensional data nor semi iso surfacing nor opaque- nor volume-rendering.

(54) and (28) both only support standard iso surfacing (which is of order $\mathcal{O}(n^{1/3})$). The here described ray tracer supports accelerated iso surfacing (which is of order $\mathcal{O}(lg(n))$), such that it outperforms those strongly especially for large data sets.

4.9 Future Research

As the here described system is mainly based on acceleration techniques that are independent of the underlying mesh it may be used as basis for other mesh ray tracers, where only the ray primitive intersection test needs to be replaced.

Chapter 5

A Visualization Framework for Time Dependent Metal Casting Simulations

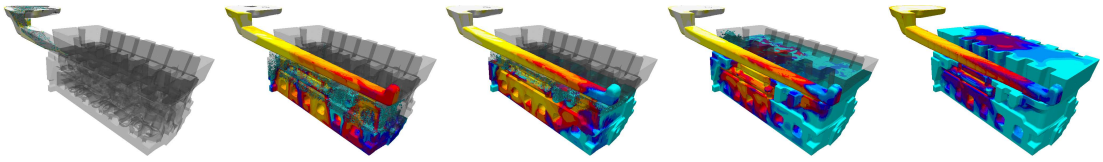

Figure 5.1: 4D filling simulation (computed using MAGMASOFT®) of a motor block. The casting mold is visualized semitransparent while the metal is displayed opaque where the different colors stand for different metal temperatures. This simulation consists of 1.2M triangles and up to 6.8M filled voxels. It is visualized on an 800×600 viewport at 14.5 fps using a dual-quadcore Xeon 1.8GHz.

5.1 Abstract

This thesis' focus is on pushing interactive CPU-based ray casting towards scientific applications. Therefore an application example has been built. It shows that CPU-based ray casting has already reached the state, where it outperforms GPU-based rasterization for some real life commercial applications.

The application example is a framework that has been built for interactively visualizing massive volumetric data generated by computer simulations of metal casting processes. Its CPU-based ray casting render kernel achieves superior visualization quality and rendering performances that has prior not been achieved by GPU-based rasterization approaches. New features as for example trilinear interpolation, switching regions of interest on/off, mapping the results correctly onto the casting mold's triangulated boundary representation, slicing the casting mold along arbitrary planes and visualizing the entire 4D metal casting simulation are all realized at interactive frame rates.

5.2 Problem

Metal casting is a manufacturing process by which molten metal is introduced into a mold, allowed to solidify and then ejected or broken out of the mold. Casting is used for creating parts of complex shape that would be difficult or uneconomical produced by other methods, such as cutting from solid material.

Metal casting big parts (e.g., ship diesel motor blocks) is rather expensive. Computer simulations are therefore used to avoid unsuccessful casts. The data generated by such computer simulations commonly exceeds several gigabytes. Interactively visualizing the full 4D-simulation is therefore challenging due to the vast amount of data provided.

For illustrating how the data to be visualized is structured, a short description of the simulation procedure is given.

One major input data is the casting mold, given as triangulated boundary representation commonly consisting of hundred of thousands of primitives. It is used as basis for automatically generating a rectilinear grid upon which the simulation is evaluated. The grid commonly consists of several millions of voxels, where 10% to 30% of them belong to the casting mold, while the rest represents the sand form. The time dependent filling- and solidification-processes are computed on this grid, where different result files are generated. The filling simulation generates for each time step metal-temperature, -pressure -velocity, air-pressure and -inclusions result files. The solidification simulation generates for each time step metal-temperatures and solidification-rate result files.

The data to be visualized consists therefore of two parts: A fine tessellated triangulation of the casting mold which is originally provided by the user, and different simulation results each consisting of several scalar fields belonging to the different time steps of the simulation.

The challenge is to visualize the massive data generated by the 4D simulation in relation to the users original triangulated boundary representation of the casting mold.

5.3 State of the Art

The literature about interactive ray tracing is vast, and a detailed description of all publications goes beyond the scope of this chapter. This section's focus is therefore only on interactive CPU-based ray tracing for large static triangulated scenes and isosurface ray tracing for large rectilinear scalar fields.

Triangle Ray Tracing

Interactive Ray tracing was first proven feasible on commodity CPUs by Wald et al. (53), using SIMD instructions on coherent ray packets for triangle intersections and traversing highly optimized *k*d-trees (13; 25). The concept of coherent traversal has been further developed, applying frustum traversal to larger packets of rays (41; 48; 52).

Isosurface Ray Tracing

Parker et al. (38; 39) employed a hierarchical grid to ray trace isosurfaces on a small supercomputer. DeMarle et al. (9) extended this implementation to clusters. Knoll et al. (22) proposed losslessly compressed octree volumes for rendering larger data. Wald et al. (50) and Groß et al. (14) used min-max *k*d-trees to ray trace isosurfaces interactively on commodity CPUs.

Isosurface Ray Tracing of Time Dependant Data

Min-max *k*d-trees have also been used for time dependent data. Marmitt et al. (28) used fast tree updates. Groß et al. (14) presented similar fast tree constructions.

5.4 Results

A visualization framework for metal casting simulations which achieves superior visualization quality and rendering performances through its CPU-based ray casting render kernel is presented here

The simulation results that are given on a rectilinear grid are shown through trilinear interpolation on the smoothened metal surface. The user may interactively (de)activate interpolation by toggling between trilinear interpolation and octree traversal. A newly introduced bit mask applied to *k*d-tree traversal allows to (de)activeate metal regions equipped with arbitrary discretized result values. Results are mapped onto the casting mold's boundary by splitting the rendering into two separate ray casting passes, where air inclusions are visualized correctly. The data set may be sliced along planes parallel to the main axes and along the arbitrary near plane. The 4D filling visualization is highly optimized by preprocessing the first ray casting pass and activating second ray casting passes in SIMD mode where applicable.

All those new features are fully interactive, allowing the user to extract crucial informations from the casting simulations fast and efficiently.

5.5 The Framework - An Overview

The framework is especially designed for metal casting simulations. A screen shot (Figure 5.2) and a short description of the framework are supplied here for giving a brief impression of its structure.

5.5.1 Visualization-, Log- and Control-Window

The visualization window displays the result types and further useful meta information. Toggle buttons on its right side are equipped with corresponding colors and result values. They are used to switch different discretized regions of interest on/off.

The log window displays failed conversion processes (due to e.g., missing files).

The control window allows the user to switch between scene control and result control. The first is for applying different features which are explained in more

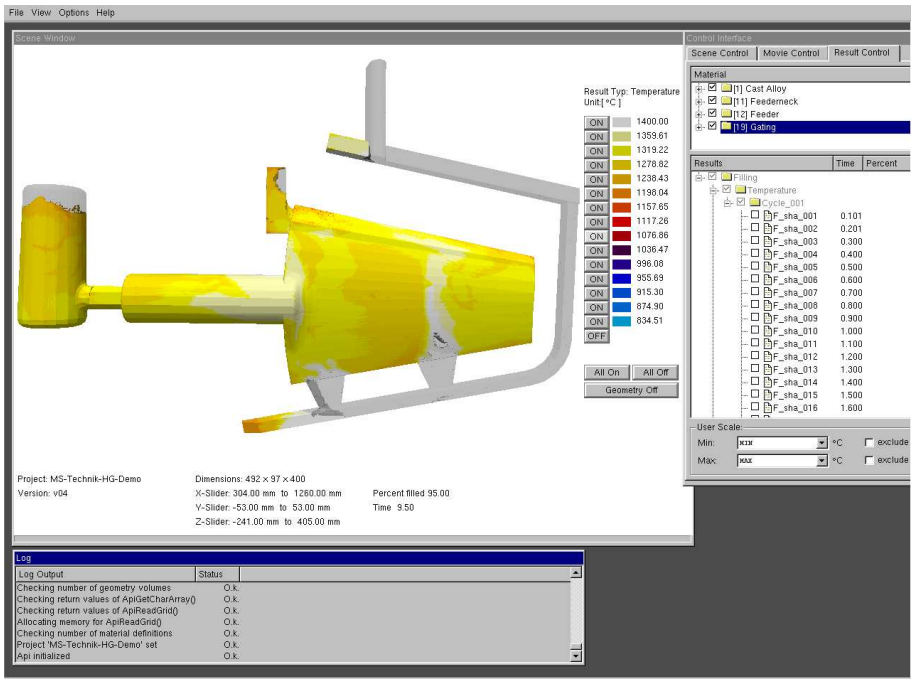

Figure 5.2: Screenshot of the framework. Left the visualization-, right the control- and bottom the log-window.

detail in the next section. The later is basically a preparation dialog, which allows to choose different casting mold parts and result types for conversion.

5.5.2 Result Conversion

The chosen casting mold parts and result types may be converted on demand. The casting mold's boundary representation is static during the filling and the solidification process. A well optimized *k*d-tree using a surface area heuristic (13; 25) and empty space clipping is therefore build in a preprocessing step. Results belonging to those casting mold parts are masked into temporary scalar fields. They are taken as basis for constructing explicit scalar *k*d-trees equipped with bit masks (see below §5.6.1) and then discarded.

Both *k*d-tree constructions are parallelized and use therefore the full potential of multi-core CPUs. The triangle *k*d-tree needs for all of the here given examples less than 30 seconds, and the scalar *k*d-trees need per time step less than 10

seconds construction time.

5.6 Realization of Different Features

The framework itself has several useful features. This sections focus is on features that evolved through the use of the CPU-based ray casting kernel. A short summary of them is given here. Their realization is later described in corresponding subsections.

1. The results and the metal boundary are visualized with trilinear interpolation. The user may interactively deactivate the interpolation in order to see the pure simulation results as voxels each equipped with a single result value.

2. Metal equipped with certain result values (e.g., solidificated metal) may be (un)interesting for the user. The user is therefore able to interactively (de)activate regions of interest.

3. The triangulated casting mold may be visualized semi-transparent. If the metal reaches the casting mold's boundary the result values are mapped onto the casting mold for displaying the results appropriately to the user's original input. Air inclusions are displayed correctly.

4. The data set may be sliced along planes parallel to the main axes and along arbitrary planes.

5. The 4D filling and the solidification process are visualized fully interactive.

These features help for efficiently extracting crucial information out of the simulation results.

5.6.1 *k*d-trees, Isosurfacing and Interpolation

The result files are given as scalar fields stored in rectilinear grids. Implicit *k*d-trees are in general a good choice for such data sets, due to their memory

overhead of only one half of the original data set size (see Chapter 3) and their fast updates/constructions (28)/(Chapter 3) for time dependent data.

The scalar data generated by casting simulations consist commonly of several million voxels, where fast updates/constructions of implicit *k*d-trees fail to achieve interactive frame rates. Furthermore do only 10% to 30% of those voxels belong to the casting mold. Explicit *k*d-trees are here a better alternative due to the relatively sparse nature of the interesting voxels and are therefore used. For each time step a corresponding tree is constructed in a pre-processing step. The resulting memory overhead is comparable to implicit *k*d-trees due to the low amount of interesting voxels. Explicit *k*d-trees have furthermore the advantage that they may be build using a surface area heuristic (13; 25) which optimizes traversal performances.

Trilinear interpolation is used for visualizing the results. A single *k*d-tree leaf node is a cuboid with result values at each of its eight corners. The result values belonging to a leaf node are discretized to 32 values which is accurate enough for correct data-interpretation, and then stored directly in the leaf node. This avoids costly memory lookups and allows for discarding the original result files from main memory.

Since many results are given only on cells partially filled with metal, two trilinear interpolations need actually to be performed. The first interpolation is for finding the metal surface. A SIMD optimized ray isosurface intersection of the Marmitt intersection test (27) is here used. The resulting isosurface is through its curved representation more accurate than the isosurface representation generated by marching cubes (24; 35). The second interpolation is for evaluating the result value at the surface point hit. Since not all eight corner points may be equipped with a result value (since there is no metal) the interpolation has to be conducted by setting the non-given corner values to zero and weighting them with zero, while the given corner values are weighted with one. The interpolated result value is then normalized by the weight.

Interpolation may also be turned off interactively, for showing the results on the rectilinear grid. The leaf node is then traversed as an octree node.

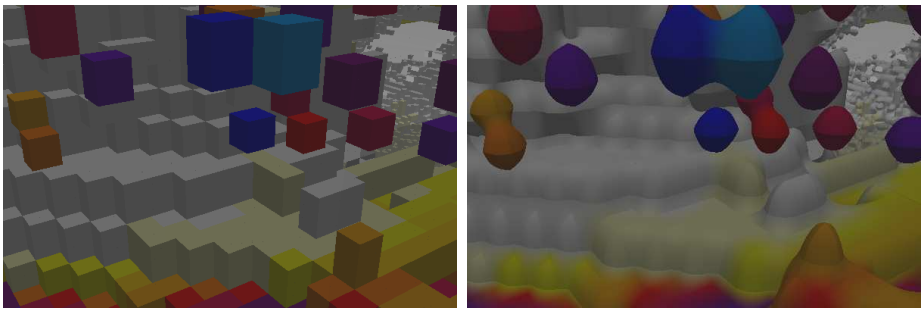

Figure 5.3: Closeup view on metal spillings with temperature as result values, once without and once with interpolation. The interpolation smoothens the metal boundary and the result values considerably. Single spills are shown in drop-like forms resulting in superiour visualization quality in comparison to octahedrons that would have been generated by marching cubes.

5.6.2 Activating / Deactivating Regions of Interest

For allowing the deactivation of metal regions equipped with certain values, it suffices to disable the ray-metal intersection tests in cells containing uninteresting result values. But this decreases the visualization performance due to the execution of unnecessary traversal steps, especially when deactivating large portions of metal. This negative effect is circumvented as follows:

The inner *k*d-tree nodes are equipped with a 32-bit mask, where each bit represents one of the 32 discretized result values. The leaf nodes' bit masks are initialized by setting the bits to either one or zero depending if the corresponding value is given in the leaf node. An inner node's bit masks is then set to the result of the *OR* operation applied on its children's bit masks. The bit masks of the entire *k*d-tree are set during the recursive tree construction, which adds only marginal overhead to this pre-processing step.

A single 32-bit mask representing which values are (de)activated for the current frame is then used during tree traversal. The sub-tree belonging to an inner *k*d-tree node is not traversed if the *AND* operation between its bit mask and the single bit mask is equal zero, since the subtree contains then only deactivated result values.

This procedure is more flexible than the traversal of min-max trees for isosur-

facing (14; 50), since it allows the (de)activation of arbitrary discretized result regions, and not only result regions greater/smaller than a given isovalue. The boundary of the activated result regions is given implicitly and is therefore not evaluated in a pre-processing step. Rays are intersected against the implicit boundary for each interactive visualization step. This stands in strong contrast to hardware accelerated rasterization, for which each boundary representation of each time step would have to be evaluated explicitly through marching cubes (24), which may take several seconds up to minutes and would create large numbers of triangles that could challenge the rasterization's interactivity.

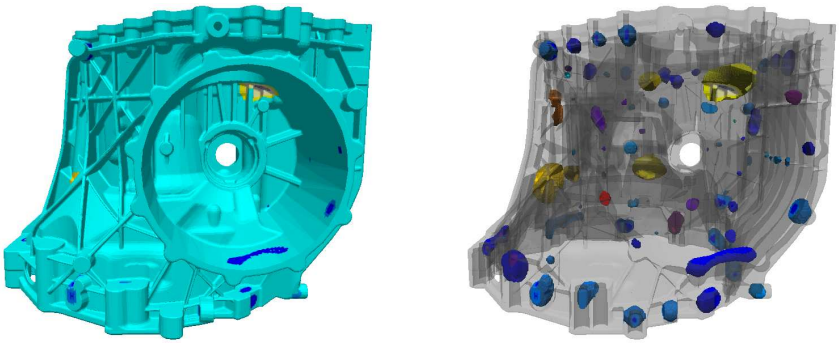

Figure 5.4: Hot spot visualization of a gear box. Left: Visualization of the temperature peaks. Right: Regions with low temperature peaks are deactivated revealing all hot spots.

5.6.3 Result Mapping onto the Triangulated Casting Mold

All simulation results are given on a rectilinear grid. The grid has automatically been generated out of the casting mold's triangulated boundary representation which was originally given by the user. The user is commonly not interested in the rectilinear grid itself, but in the results relative to the original input. An efficient interpretation of the simulation results is therefore best achieved by appropriately displaying the results in relation to the casting mold's triangulation. This is realized by mapping the results onto the triangulation at the places where the metal reaches the casting mold's boundary (Figure 5.5).

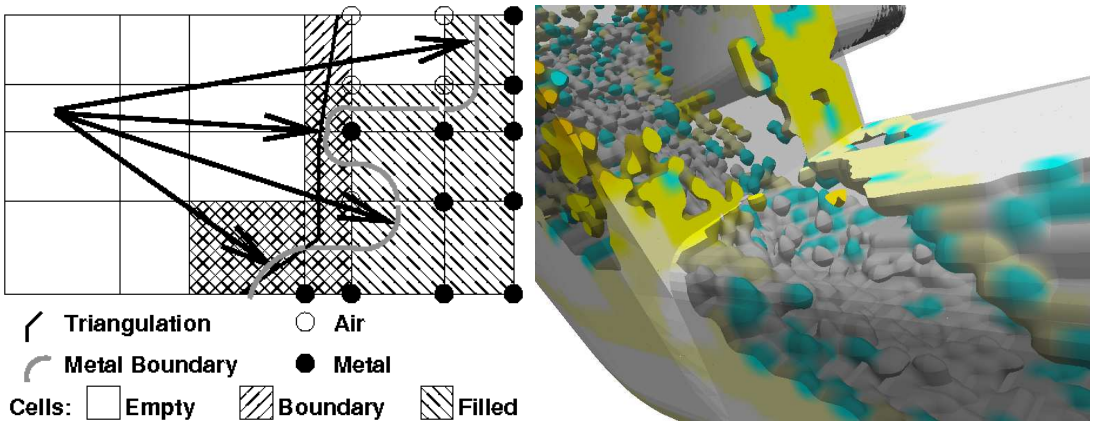

Figure 5.5: Result mapping onto the casting mold. Left: The results from the first and the third ray are not mapped onto the triangulation, since they did traverse some non-boundary cells after they previously traversed some boundary cells. Right: Result colors are shown on the triangulated boundary representation, if the metal reaches the casting mold's boundary. The result are otherwise shown on the metal surface behind the semitransparent boundary. Air inclusions are visualized correctly.

Ray casting has - unlike rasterization - the advantage that the result mapping may be executed on the fly without any pre-processing. This is achieved by splitting the rendering into two ray casting passes.

The first ray casting pass intersects the rays only against the triangulation. The normals of the intersection points define if a ray either enters or exits the casting mold. A ray's different enter- and exit-points define intervals for which the second ray casting pass is activated.

The second ray casting pass is performed on the rectilinear grid, where only metal cells of interest are ray cast. If some metal is hit, the corresponding hit normal and result values are evaluated.

Correctly mapping the results onto the geometry is challenging, since the grid representation of the casting mold is non-continuous and does not fully overlap the continuous triangulation. Simply adding the requirement that a ray has to travel a certain distance for deactivating the mapping leads to visualization artifacts, where non-existent air inclusions may be visualized, or real air inclusions may be hidden.

This problem is solved by marking the boundary cells of the casting mold's grid representation during tree construction. A result is only mapped onto the triangulation, if the corresponding ray did only traverse boundary cells. A result is not mapped, if the ray did traverse some non-boundary cells after it previously traversed some boundary cells (Figure 5.5, left).

Mapped results are visualized using the entry-triangle's normal applied with the result color. Non-mapped results are displayed by visualizing the result color on the metal surface behind the semi-transparent entry-triangle (Figure 5.5, right).

5.6.4 Clipping

Clipping along planes parallel to the main axes is in ray casting efficiently handled by clipping each ray against an axis aligned clipping box. The view frustum near plane is used for clipping along an arbitrary plane. Special care has to be taken for the two ray casting passes when combining them with clipping. Second ray casting passes have also to be executed for the two special cases where either the first triangle has been hit on the backside or where the last triangle has been hit on the front side The rays did then emerge resp. terminate inside the casting mold.

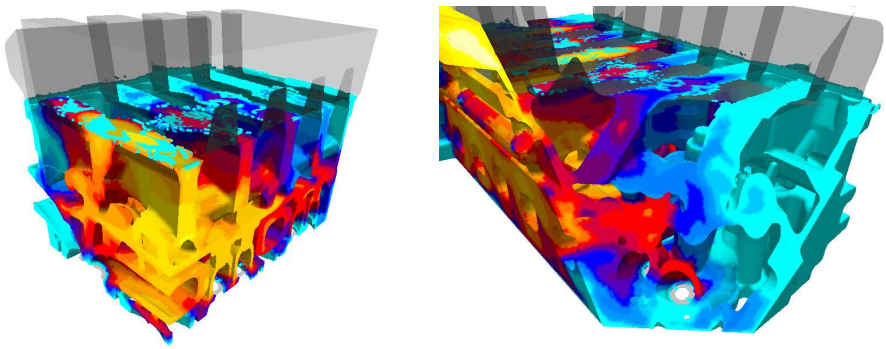

Figure 5.6: Clipping of the casting mold. Left: Using the clipping box. Right: Using the arbitrary near plane

The user may interactively change the clipping box. Clipping along an arbitrary plane is achieved by interactively changing the view frustum's near plane

and by rotating/zooming the casting mold (Figure 5.6).

5.6.5 Interactive 4D animation

The most important feature of the visualization framework is the interactive 4D animation of the casting process. Vital information is here revealed which gives clues about the success/failure of the cast.

Special implementational details have been added for reaching high visualization performances: If the 4D animation is activated, the casting process is shown repeatedly on the screen. The animation is during scene movement halted and restarted when the scene movement stops. The boundary representation of the casting mold does not change its position or form during animation, which allows to reuse the results from the first ray casting pass:

All triangle intersections are stored for each ray, where the intersection times, triangle colors and triangle normals are saved. This is done once at the animation's (re)start. For each time step of the animation the precomputed triangle intersections are used. They define the intervals for which the second ray casting pass is activated.

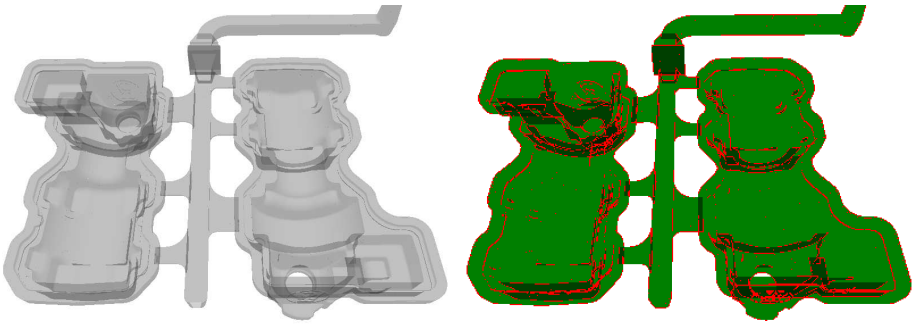

Figure 5.7: Schematic illustration of the triangle intersections pre-computation. Red: Non-SIMD friendly regions, which are ray cast as single rays. (Dark) Green: SIMD friendly regions for which (more than) one second ray casting pass is activated

If four neighboring rays have the same sequence in enter and exit points, they are SIMD friendly. The pre-computation of the triangle intersections detects such

neighboring rays and starts for those the second ray casting pass in SIMD mode (Figure 5.7). The entire pre-computation time takes less than a second which is a tolerable waiting time after stopping scene movements.

5.7 Computational Results

All results have been evaluated on a dual-quad core Xeon 1.8 GHz desktop machine. The viewport size was set to 800×600 pixels.

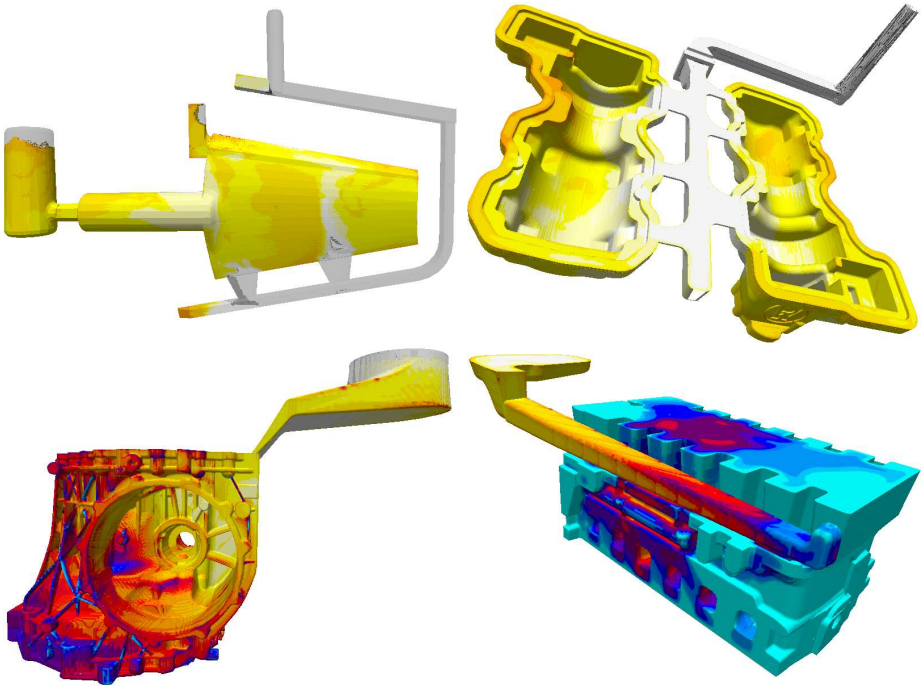

Figure 5.8: Last simulation steps of the turbine blade, pump casing, gear box and motor block datasets with metal temperature as result displayed

5.7.1 Test Data Sets

Four 4D filling simulations (computed using MAGMASOFT®) of a turbine blade, a pump casing, a gear box and a motor block are used as benchmarks (Figure 5.8). The amount of triangles of their casting mold's boundary representation,

the number of filled voxels at the last time step and the amount of time steps are given in Table 5.1.

	turbine	pump	gear	motor
triangles #	0.3M	0.5M	0.9M	1.2M
voxels #	4.2M	5.2M	5.4M	6.8M
time steps #	95	98	20	21

Table 5.1: Number of triangles, number of filled voxels at the last time step, and the amount of time steps steps for the four 4D animations.

5.7.2 Bit Mask Traversal

First the claim in 5.6.2 is verified. The bit mask applied to the *k*d-tree traversal does improve visualization speed. The four 4D animations have therefore once without and once with bit mask traversal visualized, where all metal cells have been deactivated (Table 5.2).

	turbine	pump	gear	motor
no bit mask (fps)	5.25	6.01	6.10	2.50
bit mask (fps)	73.8	37.6	25.8	17.1

Table 5.2: Performances in frames per second for non bit mask- and bit mask-traversal of the four 4D filling simulations with fully deactivated metal cells.

5.7.3 Visualization Modes

Frame rates for three different visualization modes are given in Table 5.3.

The rows in Table 5.3 stand for

1. Standard mode, applied during scene movements. Both ray casting passes are performed, both in single ray mode.

visualization mode	turbine	pump	gear	motor
standard (fps)	11.6	5.32	3.78	4.70
animation (fps)	30.3	15.6	12.7	14.5
partial-semitr. (fps)	30.4	21.5	26.3	23.3

Table 5.3: Performances in frames per second for standard, animated and partial semitransparent visualization of the four 4D filling simulations.

2. Animation mode, applied during 4D animation. The first ray casting pass has been pre-processed. The second ray casting pass is performed in SIMD mode where applicable.

3. Partial semi-transparent mode, may be applied during scene movements or 4D animation (frame rates here for 4D animation). At most one second ray casting pass is activated. The resulting image appears less disturbed since it hides complex triangulated parts behind the casting mold's front side (see Figure 5.9). Visualization performances are the same as for opaque visualization, since both visualization modes activate at most one second ray casting pass.

Figure 5.9: Full- and partial-semitransparent visualization. The user may interactively switch between both modes.

5.7.4 Discussion

5.7.4.1 Bit Mask Traversal

As claimed, the bit mask traversal outperforms non bit mask traversal, since it skips not only leaf- but also inner *k*d-tree nodes containing deactivated values. For verifying this a setup has been chosen which is most favorable for bit mask traversal. It needs to be mentioned that the performance gain is in general not that high. Bit mask traversal is very efficient if large portions of metal are deactivated. No performance differences between both traversal techniques are observed if no metal is deactivated.

5.7.4.2 Standard Mode

The standard visualization mode is barely interactive. This is due to the two ray casting passes that are performed in single ray mode. The framework therefore supplies the possibility to activate thresholded screen interpolation: The four corner rays of 4×4 pixel tiles are rendered in advance. The remaining twelve pixel colors are either evaluated via bilinear interpolation or also ray cast depending if the differences of the four resulting colors lie underneath a given threshold or not. Frame rates increase by a factor of two up to three (depending on the scene visualized and the threshold chosen) while only minor losses in visualization quality are observed.

5.7.4.3 Animation Mode

The 4D animation frame rates achieved in Table 5.3 are fairly interactive. No thresholded screen interpolation is needed here. The frame rates are roughly three times as high as in standard mode, due to the saving of the first ray casting pass and the SIMD support.

5.7.4.4 Partial Semi-Transparent Mode

The partial semi-transparent mode is especially fast for complex scenes, since many ray casting passes may be saved. It achieves highly interactive frame rates during 4D animation.

5.7.4.5 Turbine Blade

Interesting are the frame rates achieved for the turbine blade. Due to its special form and visualization (see Figure 5.8) are second ray casting passes almost never performed more than once. This leads to superior frame rates for bit mask traversal (Table 5.2), but also to no further frame rate improvements when switching from fully semitransparent- to partial semitransparent-visualization (Table 5.3).

5.8 Future Research

The ray casting based render kernel of this visualization framework is suited for further development, where it may support the visualization of computational results from other fluid dynamics simulations.

Chapter 6

Summary and Conclusions

In this final chapter the contents and all new contributions of this thesis are briefly summarized.

Chapter 1 gives the motivation for this thesis. First a brief overview to different rendering techniques and methods, acceleration techniques for ray tracing, and hierarchical acceleration structures is given. The complexity analysis for kd-trees and skd-trees shows that if a reasonable scene of n primitives is equipped with a (s)kd-tree as acceleration structure the visualization time is of order $\mathcal{O}(lg(n))$. As those trees have first to be constructed (construction times between $\mathcal{O}(n)$ and $\mathcal{O}(lg(n))$) it is most reasonable to use interactive ray casting on large static scenes, where the trees are constructed only once when loading the data set. For large static data sets ray casting with the strongly sublinear visualization time of order $\mathcal{O}(lg(n))$ will outperform rasterization which only has a liner visualization time of order $\mathcal{O}(n)$.

Chapter 2 introduces a new OpenGL friendly interface that allows the simple and fully interactive integration of different ray casters into already existing OpenGL frameworks or vice versa it supports to render primitives drawn by the graphics card into ray cast scenes. The interface also allows to run the CPU-based ray casters and the GPU-based rasterization in parallel and furthermore supports shadows. The technique allows to combine the advantages of today's ray casters and graphics cards.

Chapter 3 introduces a new and very general definition for implicit kd-trees together with their construction and traversal algorithms. New implicit bit, bit-

mask, and min/max kd-trees are defined. An efficient memory reduction scheme for (implicit) min/max kd-trees has been introduced. The resulting optimized implicit min/max or max kd-trees require as much resp. half as much memory as the original scalar field. They allow on today's desktop computers interactive iso surfacing and MIP of data sets that are bigger than one half of the machine's main memory. It has furthermore been shown that the construction of implicit kd-trees is linear, scales linear to the amount of processor cores used, and is fairly fast.

Chapter 4 introduces an interactive for general SIMD operations optimized ray tracing technique for large tetrahedral meshes that scales for static scenes sublinear to data size. Compared to other ray tracing systems this system achieves framerates that are comparable to the best render times in volume- and iso surface-rendering of those systems, where this system is additionally the most portable and by far the most general, such that it is best suited for portation upon different (future) hardware and for usage upon several applications.

Chapter 5 shows as application example a framework that has been built for interactively visualizing the massive volumetric data generated by computer simulations of metal casting processes. Its CPU-based ray casting render kernel achieves superior visualization quality and rendering performances that has prior not been achieved by GPU-based rasterization approaches.

Appendix A

Curriculum Vitae

Dipl.-Math. techn. Matthias Groß

E-Mail: matthias.s.gross@web.de

Date of Birth: 26. December 1979

Place of Birth: Neustadt an der Weinstraße (Germany)

Nationality: German

Education

2005 – 2009	PhD student at Fraunhofer Institut für Techno- und Wirtschaftsmathematik (institute for industrial and business mathematics, ITWM): competence center high performance computing and visualization (HPC) University of Kaiserslautern (TU-KL): department of computer sciences, computer graphics group
2000 – 2005	German diplom (corresponds to master of science) in industrial mathematics at TU-KL, diploma thesis: “Interactive Ray Casting for Rectilinear Grid Aligned Data Sets”
1990 – 2000	German abitur (corresponds to general qualification for university entrance) at Käthe-Kollwitz-Gymnasium, Neustadt/Wstr.
1996 – 1997	exchange student at West High, Manchester/NH (USA)

Employment History

2009 – now	software developer at the Schlumberger company Integrated Exploration Systems (IES)
2004 – 2008	research assistant at ITWM/HPC
2002 – 2005	course instructor and tutor for C/C++ classes at TU-KL

Languages & Professional Qualifications

German:	native speaker
English:	fluent in spoken, written and business English
expert in:	C/C++ (platform independant for both Windows and Linux)
experienced in:	Qt, Open Inventor, OpenGL, SSE, Posix Threads; LaTeX

Scholarships & Awards

2005	Fraunhofer PhD-scholarship (exceptional raise in 2007)
1999 & 2000	in each of both years awardee of the first and second round of the German national competition in mathematics
2000	DPG-book award and an annual free membership of the "Deutsche Physikalische Gesellschaft" (German physics association) for the excellent achievements in the school subject physics

Publications

2008	Interactive SIMD Ray Tracing for Large Deformable Tetrahedral Meshes Matthias Groß, Hans Hagen and Franz-Josef Pfreundt RT08: IEEE Symposium on Interactive Ray Tracing
2007	A Visualization Framework for Time Dependent Metal Casting Simulations Matthias Groß, Carsten Lojewski and Hans Hagen RT07: IEEE Symposium on Interactive Ray Tracing

2007	Fast Implicit *k*d-Trees: Accelerated Isosurface Ray Tracing and Maximum Intensity Projection for Large Scalar Fields Matthias Groß, Carsten Lojewski, Martin Bertram and Hans Hagen CGIM07: Proceedings of Computer Graphics and Imaging

Part-Time Jobs & Activities

2003 & 2004	extension of the since 2002 voluntary mentoring-commitment for the mathematics international students of the TU-KL to a part-time job
1988 - 1998	member of the German boyscouts
sports:	different kinds of, currently badminton
games:	board games

June 27, 2009

References

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference Proceedings*, volume 32, pages 37–45, 1968. 2

[2] Daniel A. Balciunas, Lucas P. Dulley, and Marcelo K. Zuffo. Gpu-assisted ray casting of large scenes. In *In Proceedings of IEEE Symposium on Interactive Ray Tracing '06*, pages 95–103, sept 2006. 9

[3] Marcel Beister, Manfred Ernst, and Marc Stamminger. A hybrid gpu-cpu renderer. In *Proceedings of Vision, Modeling and Visualization*, pages 415–420, 2005. 9

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. 5

[5] Brian C. Budge, Daniel Coming, Derek Norpchen, and Kenneth I. Joy. Accelerated building and ray tracing of restricted bsp trees. *IEEE Symposium on Interactive Ray Tracing RT08.*, pages 167–174, Aug. 2008. 5

[6] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005. 59

[7] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976. 6

[8] David E. DeMarle, Christiaan P. Gribble, and Steven G. Parker. Memory-savvy distributed interactive ray tracing. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, 2004. 33

[9] David E DeMarle, Steven Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed interactive ray tracing for large volume visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 87–94, 2003. 33, 89

[10] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, 1980. 4

[11] Michael P. Garrity. Raytracing irregular volume data. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 35–40. ACM, 1990. 59

[12] Joachim Georgii and Rudiger Westermann. A generic and scalable pipeline for gpu tetrahedral grid rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1345–1352, 2006. 59

[13] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987. 5, 10, 89, 91, 93

[14] Matthias Groß, Carsten Lojewski, Martin Bertram, and Hans Hagen. Fast implicit kd-trees: Accelerated isosurface ray tracing and maximum intensity projection for large scalar fields. In *Proceedings of Computer Graphics and Imaging (CGIM07)*, pages 67–74, june 2007. 62, 89, 95

[15] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech. Technical University Prague, 2001. 4, 5

[16] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On the fast construction of spatial hierarchies for ray tracing. In *IEEE Symposium on Interactive Ray Tracing RT06*, pages 71–80, 2006. 6

[17] Warrent Hunt, William R. Mark, and Gordon Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing RT06*, pages 81–88, 2006. 5

[18] Thiago Ize, Ingo Wald, and Steven G. Parker. Asynchronous bvh construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2007. 6

[19] Thiago Ize, Ingo Wald, and Steven G. Parker. Ray tracing with the bsp tree. In *IEEE Symposium on Interactive Ray Tracing, RT08*, pages 159–166, 2008. 5

[20] Ravi P. Kammaje and Benjamin Mora. A study of restricted bsp trees for ray tracing. In *IEEE Symposium on Interactive Ray Tracing, RT07*, pages 55–62, 2007. 5

[21] C. E. Kim. Three-dimensional digital planes. In *IEEE Transaction on Pattern Analysis and Machine Intelligence PAMI-6*, pages 639–645, 1984. 1

[22] Aaron Knoll, Ingo Wald, Steven G Parker, and Charles D Hansen. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 115–124, 2006. 89

[23] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, 1994. 2

[24] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 163–169, July 1987. 93, 95

[25] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, 1990. 5, 10, 89, 91, 93

[26] Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek. Recent Advancements in Ray-Tracing based Volume Rendering Techniques. In *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005*, pages 131–138, November 2005. 59

[27] Gerd Marmitt, Andreas Kleer, Ingo Wald, Heiko Friedrich, and Philipp Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Vision, Modelling, and Visualization 2004 (VMV)*, pages 429–435. Akademische Verlagsgesellschaft Aka, 2004. 53, 93

[28] Gerd Marmitt and Philipp Slusallek. Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS) 2006*, pages 235–242, May 2006. 59, 76, 85, 86, 89, 93

[29] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 27–33, 1990. 1

[30] Benjamin Mora and David S. Ebert. Low-complexity maximum intensity projection. *ACM Trans. Graph.*, 24(4):1392–1416, 2005. 33, 55

[31] Benjamin Mora, Jean-Pierre Jessel, and René Caubet. A new object-order ray-casting algorithm. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 203–210, 2002. 33

[32] Lukas Mroz, Helwig Hauser, and Eduard Gröller. Interactive high-quality maximum intensity projection. In *In Proceedings of EUROGRAPHICS*, pages 341–350, 2000. 33

[33] Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *In Proceedings of BRL-CAD Symposium '95*, pages 93–100, june 1995. 10

[34] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993. 13

[35] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 83–91. IEEE Computer Society Press, 1991. 93

[36] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Spatial kd-tree: An indexing mechanism for spatial databases. In *In Proceedings of the IEEE COMPSAC Conference*, 1987. 6

[37] S. Parker, M. Parker, Y. Livnat, P.P. Sloan, C.D. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, July-September 1999. 5, 59

[38] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization*, pages 233–238, October 1998. 33, 89

[39] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing. *Interactive 3D Graphics (I3D)*, pages 119–126, april 1999. 10, 33, 50, 54, 89

[40] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Phillip Slusallek. Experiences with streaming construction of sah kd-trees. In *IEEE Symposium on Interactive Ray Tracing RT06*, pages 89–94, 2006. 5

[41] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level Ray Tracing Algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)*, 24(3):1176–1185, 2005. 89

[42] C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon. GPU-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing (RITA)*, 12(2):9–29, 2005. 59

[43] Marc Stamminger, Jörg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective textures. In *Proceedings of the 11th EUROGRAPHICS Workshop on Rendering*, pages 377–388, 2000. 9

[44] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In T. Akenine-Möller and W. Heidrich, editors, *Rendering Techniques 2006 (Proc. of 17th Eurographics Symposium on Rendering)*, pages 139–149, 2006. 10

[45] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding intervall hierarchy. In *IEEE Proceedings of the Eurographics Symposium on Rendering*, pages 139–149, 2006. 6

[46] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 10, 39

[47] Ingo Wald. On fast construction of sah based bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing RT07*, pages 33–44, 2007. 6

[48] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):6, 2007. 6, 10, 89

[49] Ingo Wald, Heiko Friedrich, Aaron Knoll, and Charles D Hansen. Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes. *IEEE Transactions on Visualization and Computer Graphics*, pages 1727–1734, 2007. 59, 85, 86

[50] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Phillip Slusallek, and Hans Peter Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, sept/oct 2005. 33, 35, 45, 53, 54, 89, 95

[51] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *IEEE Symposium on Interactive Ray Tracing RT06*, pages 61–69, 2006. 5

[52] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, pages 485–493, 2006. (Proceedings of SIGGRAPH 2006). 5, 10, 89

[53] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In S.J.Gortler and K.Myszkowski, editors, *Rendering Techniques 2001 (Proceedings of the 12th EUROGRAPHICS Workshop on Rendering*, pages 277–288. Springer, 2001. 89

[54] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization '03*, pages 333–340. IEEE, 2003. 60, 70, 72, 76, 85, 86

[55] Manfred Weiler, Paula N. Mallon, Martin Kraus, and Thomas Ertl. Texture-encoded tetrahedral strips. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, pages 71–78. IEEE Computer Society, 2004. 60, 85

[56] Lee Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, 1990. 2

[57] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980. 2

[58] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Comput. Graph. Appl.*, 10(6):13–32, 1990. 20

[59] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, pages 67–77, 2006. 6